МІНІСТЕРСТОВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
КАФЕДРА КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____ А.С. Савченко
«____» _____ 2020 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ МАГІСТРА

ЗА СПЕЦІАЛЬНІСТЮ 122

«КОМП'ЮТЕРНІ НАУКИ ТА ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ»

Тема: «Система контейнеризації мікросервісів на базі Docker»

Виконавець: студент УС-211м Московенко Євгеній Олегович
(студент, група, прізвище, ім'я, по батькові)

Керівник: д.т.н., проф. Зіатдінов Юрій Кашафович
(науковій ступень, вчене звання, прізвище, ім'я, по батькові)

Нормоконтролер _____ І.Е. Райчев
(підпис) (П.І.Б.)

КИЇВ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютерних інформаційних технологій
Освітній ступінь: Магістр
Напрям: 122 «Комп'ютерні науки та інформаційні технології»
(шифр, найменування)

ЗАТВЕРДЖУЮ
Завідувач випускової кафедри
_____ А.С. Савченко
«____» _____ 2020 р.

ЗАВДАННЯ
на виконання дипломної роботи студента
Московенко Євгенія Олеговича
(П.І.Б. випускника)

1. Тема роботи «Система контейнеризації мікросервісів на базі Docker» затверджена наказом ректора від 25.09.2019 р. № 2175/ст, № пор. 5.

2. Термін виконання роботи: з 14.10.2019 р. по 09.02.2020 р.

3. Вихідні дані роботи: тема, перелік літератури, календарний план-графік написання дипломної роботи.

4. Зміст пояснювальної записки:

Вступ
Розділ 1 Особливості побудови веб-додатків
Розділ 2 Аналіз мікросервісної архітектури
Розділ 3 Docker як засіб контейнеризації мікросервісів
Розділ 4 Розробка системи контейнеризації мікросервісів
Висновки
Список бібліографічних посилань використаних джерел
Додаток А
Додаток Б
Додаток В

5. Перелік обов'язкового ілюстративного матеріалу: рисунки.

6. Календарний план-графік:

№ з/п	Завдання	Термін виконання	Підпис керівника
1	Аналіз і опрацювання літератури	14.10.2019р. – 20.10.2019р.	
2	Провести консультацію з науковим керівником щодо розділів дипломної роботи	20.10.2019р.	
3	Підготовка та написання розділу 1	20.10.2019р. – 30.10.2019р.	
4	Підготовка та написання розділу 2	30.10.2019р. – 15.11.2019р	
5	Підготовка та написання розділу 3	15.11.2019р. – 30.11.2019р.	
6	Підготовка та написання розділу 4	30.11.2019р. – 20.12.2019р.	
7	Оформлення пояснювальної записки	20.12.2019р. – 05.01.2020р.	
8	Оформлення графічної частини роботи	05.01.2020р. – 15.01.2019р.	
9	Подати дипломну роботу керівнику	15.01.2019р.	
10	Підготовка до захисту дипломної роботи	15.01.2020р. – 09.02.2020р.	

7. Дата видачі завдання: 14.10.2019 р.

Керівник дипломної роботи: _____ Зіатдінов Ю. К.
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання: _____ Московенко Є.О.
(підпис випускника) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Система контейнеризації мікросервісів на базі Docker». Робота містить 44 рисунки та 4 таблиці.

КОНТЕЙНЕРИЗАЦІЯ, МІКРОСЕРВІС, DOCKER, КЛАСТЕРИЗАЦІЯ, ОРКЕСТРАЦІЯ.

Об'єкт дослідження – технологія контейнеризації.

Мета дипломної роботи – розробка мікросервісної архітектури, кластеризація, оркестрація, моніторинг на базі Docker.

Метод дослідження – дослідження існуючих концепцій проектування мікросервісної архітектури.

Область застосування – веб-додатки.

У процесі роботи над дипломною роботою, було зроблено дослідження принципів побудови мікросервісної архітектури, проаналізовано основні можливості і функції системи контейнеризації Docker.

Було спроектовано масштабовану відмовостійку кластерну інфраструктуру мікросервісів на базі Docker. Додатково було реалізовано концепцію безперервної доставки, стандартизовано формат взаємодії між сервісами, налаштовано моніторинг системи.

Результати: створена інфраструктура може бути застосована для реалізації комерційних продуктів. На базі створеної інфраструктури можна побудувати продукт із мінімальними оновленнями архітектури.

Дану роботу можна використовувати у якості методичного матеріалу розробникам ПЗ при проектуванні і розробці мікросервісних систем.

Розробка та дослідження проводилися під управлінням ОС Windows 10 у інтегрованому середовищі розробки PHPSTORM, мова програмування – PHP.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

DDD	—	Domain Driven Design – предметно-орієнтована архітектура
ОС	—	Операційна система
IDE	—	Integrated development environment – інтегроване середовище розробки
ПЗ	—	Програмне забезпечення
CI	—	Continuous Integration – безперервна інтеграція
CD	—	Continuous Delivery – безперервна доставка
VM	—	Virtual Machine – віртуальна машина
LB	—	Load Balancer – балансер навантаження
LXC	—	Linux Containers – контейнери Linux
REST	—	Representational State Transfer – передача репрезентативного стану
API	—	Application Programming Interface – прикладний програмний інтерфейс

ВСТУП

Мікросервісна архітектура або мікросервіси – це метод розробки програмного забезпечення, за допомогою якого зосереджується увага на побудові однофункціональних модулів (сервісів) з чітко визначеними завданнями та операціями.

Важливо розуміти, що під сервісом розуміється цілий набір послуг і визначений функціонал, який надають споживачеві. А мікросервіси – це дроблення функціоналу так, щоб він був легко замінюваним і доступним іншим частинам системи.

Кожен мікросервіс – це невелика монолітна програма, яка виконує свою функцію. У програмний продукт при розробці за допомогою мікросервісної архітектури можна додавати будь-яку кількість нових мікросервісів, розширюючи його функціональність. Щоб домогтися подібного в монолітної програмі, необхідно вносити зміни в основний продукт.

Мікросервісна архітектура – це створення розподілених додатків у вигляді набору невеликих незалежних сервісів, які окремо розробляються та розгортаються, і можуть запускатись як один або кілька ізольованих процесів.

Додатковим плюсом використання мікросервісів для реалізації бізнес-функцій є їх однакова робота в різних системах. Наприклад, використовуючи в різних системах один і той же мікросервіс, що видає паспортні дані клієнта за запитом, всюди отримаємо ідентичний результат в однаковому вигляді. Це є гарантією стабільної якості роботи ПЗ.

Переваги мікросервісів з точки зору розробки:

- об’єм програмного коду є мінімальним – сервіс не повинен вимагати великої кількості людей для розробки. Одна команда може розробляти кілька сервісів;
- слабка зв’язність – зміни в одному сервісі не впливають на інший;

- модульність – кожен мікросервіс націлений на вирішення лише однієї групи задач;
- висока доступність – якщо якась частина моноліту вийде з ладу – зламається весь додаток. У випадку із мікросервісами, не всі компоненти можуть працювати (на кшталт авторизація), але додаток при цьому залишиться доступним;
- використання різноманітних технологій – при розробці кожного сервісу розробники вільні вибирати інструменти, які найкраще підійдуть для конкретної бізнес-логіки у даному сервісі. Наприклад, можна вибрати оптимальну базу даних і зручні інструменти для роботи з нею. Мікросервісна архітектура також дозволяє спробувати нову технологію на окремому сервісі, не переписуючи при цьому весь додаток.

Недоліки мікросервісів з точки зору розробки:

- складність розробки – розробка з використанням мікросервісів є довготривалим процесом, що не підходить, коли потрібне швидке рішення (прототип, невеликий додаток, стислі терміни);
- складність підтримки – кожен мікросервіс потребує окремого обслуговування, тому потрібен постійний автоматизований моніторинг та логування даних.

Контейнеризація – це «легка» віртуалізація і ізоляція ресурсів на рівні операційної системи, яка дозволяє запускати додаток і необхідний йому мінімум системних бібліотек в повністю стандартизованому контейнері, що з'єднується з хостом за допомогою мережевих інтерфейсів. Контейнер не залежить від ресурсів або архітектури хоста, на якому він працює.

Всі компоненти, необхідні для запуску програми, упаковуються як один образ і можуть бути використані повторно. Додаток в контейнері працює в ізольованому середовищі і не використовує пам'ять, процесор або диск

хостовой операційної системи. Це гарантує ізольованість процесів всередині контейнера.

Docker – це програмне забезпечення для автоматизації розгортання і управління додатками в середовищах з підтримкою контейнеризації.

В основі роботи Docker лежить стандартизований спосіб виконання коду. Docker – це операційна система для контейнерів. Подібно до того як віртуальна машина створює віртуальне представлення апаратного забезпечення сервера (тобто усуває необхідність безпосередньо управляти ним), контейнери створюють віртуальне представлення серверної операційної системи. Після установки на кожен сервер, Docker надає доступ до простих команд, необхідним для збірки, запуску або зупинки контейнерів.

Зазвичай мікросервісна архітектура застосовується як один з варіантів масштабування додатків, а саме:

- Sharding (*розбиття*) – дані і інструменти для доступу і обробки до них розміщуються на різних вузлах;
- Mirroring (*створення дзеркал*) – дублювання даних на декількох однакових вузлах;
- Модульність – функціональність розбита по декільком сервісам, при чому кожен сервіс може бути створений своїми засобами розробки, написаний на різних мовах програмування.

В останні роки тенденція використання мікросервісів набула популярності, коли підприємства прагнуть, щоб їх програмне забезпечення було більш гнучким та масштабованим. У мікросервісній архітектурі додаток розбивається на колекцію слабко пов'язаних служб.

Використовуючи мікросервісний підхід можна: збільшити доступність бізнес-додатків в кілька разів; протягом тижня встановлювати десятки оновлень, не ризикуючи порушити працездатність корпоративної

інформаційної системи; розробляти оновлення паралельно силами кількох команд, що створюють нові версії незалежно один від одного.

Впровадження мікросервісів дає наступні переваги бізнесу:

- оперативне розширення і вдосконалення функцій без порушення інших мікросервісів;
- тестування нових функцій без порушення робочих – це можливо завдяки заміні або додаванню мікросервісів в систему;
- безпечне впровадження інновацій;
- взаємозамінність модулів;
- готовність до горизонтального і вертикального масштабування;
- безперебійна робота всіх компонентів і системи в цілому навіть при відмові ряду мікросервісів;
- централізоване внесення змін.

Використання Docker дозволяє швидше і ефективніше доставляти або переміщати код, стандартизує виконуються додатками операції і в цілому економить кошти, оптимізуючи використання ресурсів. Завдяки Docker користувачі отримують об'єкт, який з високою надійністю можна запускати на будь-якій платформі. Простий і зрозумілий синтаксис Docker забезпечує повний контроль над виконуваними операціями.

Docker контейнери є ізольованими, що є величезною перевагою з точки зору безпеки та автономності сервісів.

Контейнери Docker можна використовувати в якості основних компонентів для створення сучасних платформ і додатків. Docker спрощує збірку і запуск розподілених мікросервісних архітектур, розгортання коду за допомогою стандартизованих конвеєрів безперервної інтеграції і доставки, створення добре масштабованих систем обробки даних і повністю керованих платформ для розробників.

Docker складається з двох процесів:

- демона Docker, який запускається на гостьовій машині або всередині VirtualBox;
- клієнта, через який можна взаємодіяти з демоном.

Образ Docker (*Docker image*) – містить операційну систему, застосунок і всі його залежності. Образи в Docker складаються з шарів. Наприклад, якщо потрібно образ з веб-сервером, можна взяти за основу образ з дистрибутивом операційної системи, додати залежність – веб-сервер, і створити новий образ, який матиме два шари – один з ОС, наступний з веб-сервером. Образами можна обмінюватись через DockerHub.

DockerHub являє собою хмарний сервіс для організації спільної роботи, автоматизації робочого процесу, створення, поширення і запуску адаптованих для Docker застосунків. По суті DockerHub надає набір сервісів, таких як поширення образу контейнера, управління змінами, організація взаємодії між користувачами і розробниками, супровід життєвого циклу, інтеграція зі сторонніми службами.

Модель монетизації сервісу DockerHub аналогічна GitHub — робота над публічними проектами безкоштовна і лише при необхідності використання приватних репозиторіїв стягується оплата.

Контейнер Docker – це запущений образ. Контейнери Docker можна запускати, спиняти, переміщувати і видаляти. Також можна зробити «commit» (*знімок*) контейнера, що створить образ з поточного стану контейнера.

Отже, для великих додатків корпоративного масштабу використання мікросервісної архітектури та Docker рішень – оптимальний варіант, що дозволяє швидко реагувати на зміну ринку.

РОЗДІЛ 1
ОСОБЛИВОСТІ ПОБУДОВИ ВЕБ-ДОДАТКІВ

1.1. Характеристика веб-додатків

Веб-додаток – клієнт-серверний додаток, у якому клієнт взаємодіє з веб-серверами за допомогою браузера. Логіка веб-додатків розбита між серверами та клієнтами. Дані зберігаються та оброблюються на серверах, до яких можна отримати доступ за допомогою браузера, мобільного додатку і тд, через протоколи HTTP або HTTPS (протокол із шифруванням інформації на базі TLS).

Веб-додаток схожий звичайний комп'ютерний застосунок, за винятком того, що він працює через Інтернет.

Архітектура веб-додатків описує взаємодію між додатками, базами даних та іншими проміжними компонентами.

Веб-архітектура – це концептуальна структура Всесвітньої павутини. Всесвітня павутина або інтернет дозволяє спілкуватися між різними користувачами та між різними системами та підсистемами.

Основою цього є різні компоненти та формати даних, які формують інфраструктуру Інтернету, що стає можливим завдяки трьом основним компонентам протоколів передачі даних (TCP / IP, HTTP, HTTPS), форматах представлення (HTML, CSS, XML, JSON, YAML) та стандартам адресації (URI, URL).

Термін веб-архітектура слід відрізняти від термінів архітектура веб-сайтів та архітектура інформації.

У будь-якому типовому веб-додатку є два компоненти (рис.1.1):

- програмний код на стороні клієнта (*frontend складова*)
- програмний код на стороні сервера (*backend складова*)

Програмний код на стороні клієнта реагує на події від користувача, такі як ввід, натискання на кнопку, наведення курсору на елемент. Комбінація CSS, HTML та JavaScript використовується для написання коду на стороні клієнта. Цей код аналізується веб-браузером. На відміну від серверного коду, клієнтський код можна переглянути, а також змінювати.

Клієнтський код зв’язується з веб-сервером лише через HTTP-запити і не може безпосередньо читати файли або дані з сервера.

Програмний код на стороні сервера виконує задачі по обробленню, зберіганню, агрегації даних. Для написання коду на стороні сервера використовуються такі високорівневі мови програмування як C#, Java, Python, PHP, Ruby. Серверна частина відповідає на HTTP-запити, відтворює сторінку, яку запитував користувач, обробляє дані, які надіслав користувач.

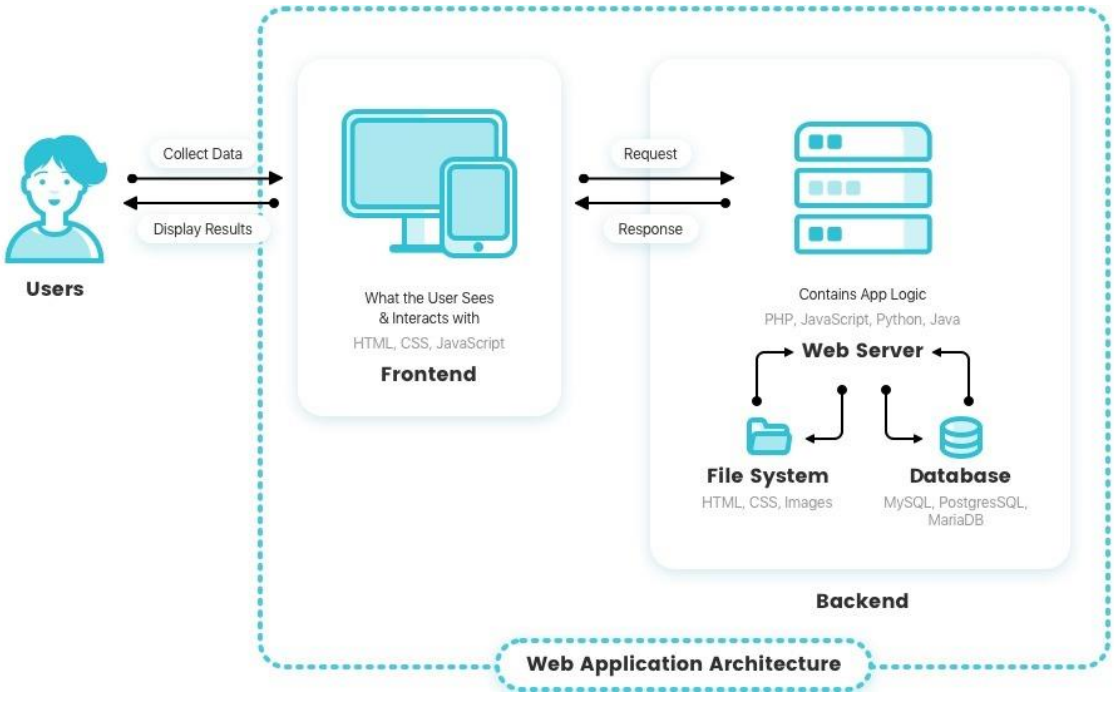


Рис.1.1 Взаємодія користувача із веб-додатком

1.2. Основі компоненти веб-інфраструктури

HTTP – це протокол, який дозволяє отримувати різні ресурси, наприклад HTML-документи, картинки, css-файли і тд. Протокол HTTP лежить в основі обміну даними в мережі Інтернет.

HTTP є протоколом клієнт-серверного взаємодії, що означає ініціювання запрошення до серверів самостійно отримувачем, зазвичай це веб-браузер.

Клієнти та сервери взаємодіють, обмінюються однорідними повідомленнями (а не потоком даних). Повідомлення, які відправлені клієнтом, називають запитом, а повідомлення, що відправляє сервер – відповідь.

Між цими запитами і відповідями як правило існують численні посередники, так звані проксі (*proxy*), які виконують різні операції і працюють як шлюзи або кеш (рис. 1.2).

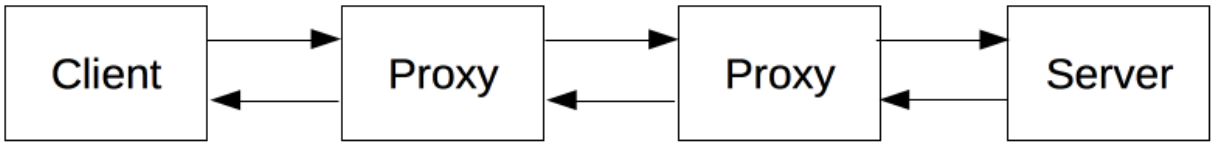


Рис. 1.2. Посередники між клієнтом і сервером

Зв'язок між клієнтами та серверами здійснюється за допомогою запитів та відповідей (рис. 1.3):

1. Клієнт (браузер) надсилає HTTP запит;
2. Веб-сервер отримує запит;
3. Сервер запускає додаток для обробки запиту;
4. Сервер повертає HTTP відповідь браузеру;
5. Клієнт (браузер) отримує відповідь;

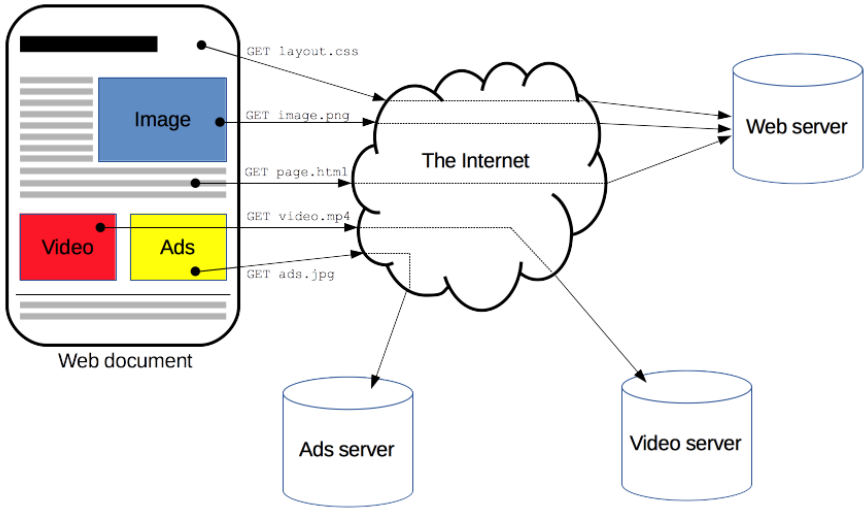


Рис. 1.3. Взаємодія клієнта і сервера

HTTP є протоколом прикладного рівня, який використовує можливості іншого протоколу – TCP (або TLS – захищений TCP) – для пересилки своїх повідомлень (рис. 1.4). Проте будь-який інший надійний транспортний протокол теоретично може бути використаний для доставки таких повідомлень.

Завдяки власній розширюваності, протокол HTTP використовується не тільки для пересилки гіпертекстових документів, зображені та відео, але і для передачі файлів, наприклад, за допомогою HTML-форми.

HTTP також може бути використаний для отримання тільки частини документа (наприклад, AJAX запит).

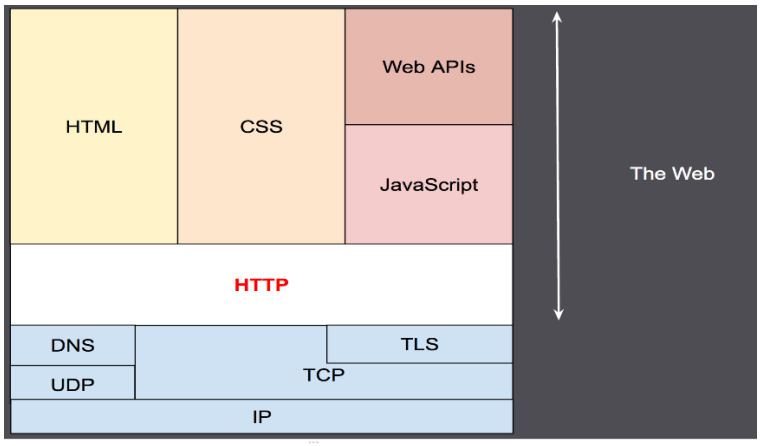


Рис. 1.4. Складові HTTP протоколу

На рис. 1.5 зображено приклад взаємодії клієнта із веб-сервером і його компонентами.

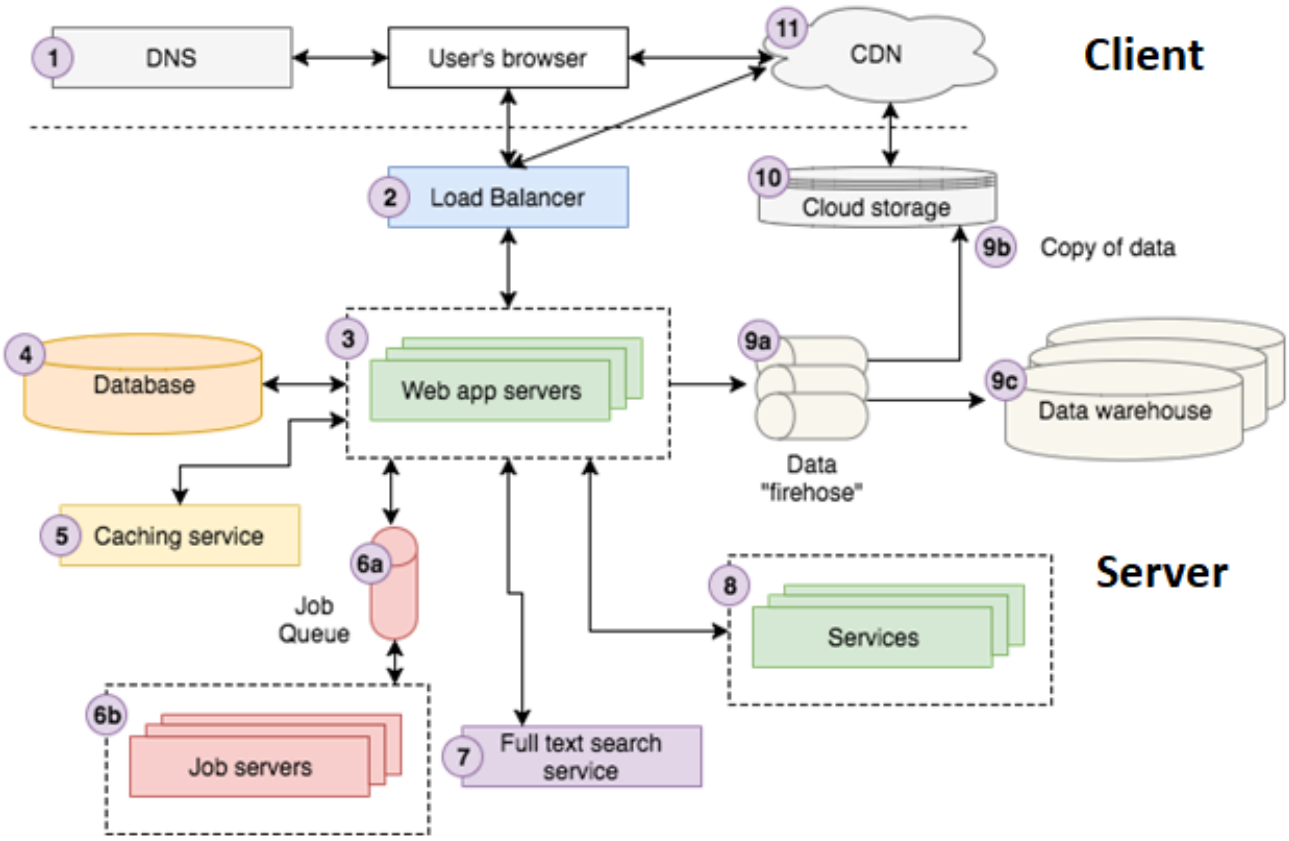


Рис. 1.5. Взаємодія клієнта із веб-сервером і його компонентами

Розглянемо поетапно, як клієнт взаємодіє із веб-сервером.

Користувач, використовуючи веб-браузер, виконує пошук інформації. Коли користувач натискає на посилання, браузер перенаправляє на сторінку веб-сайту. Браузер користувача надсилає запит на DNS-сервер (**1**), щоб дізнатися, як зв'язатися із веб-сайтом, тобто знайти ip адресу веб-сервера, використовуючи доменне ім'я, і потім надсилає запит.

Запит приходить до балансера навантаження (*load balancer, LB*) (**2**), який випадковим чином обирає один із декількох веб-серверів (**3**), які задіяні для обробки запитів.

Веб-сервер шукає інформацію у базі даних (**4**) та отримує деякі дані із служби кешування (*caching service*) (**5**).

Далі веб-сервер відправляє задачу (*job*) до черги завдань (**6a**), яку «сервери роботи» (*job servers*) (**6b**) оброблять асинхронно. Наприклад, це може бути відправка email листа або обробка відео-файлу, що може зайняти деякий час.

Веб-сервер відправляє запит до служби повнотекстового пошуку (*full text search service*) (**7**), щоб швидко знайти потрібні дані із великого обсягу інформації.

Далі веб-сервер взаємодіє із допоміжними сервісами (**8**), які надають потрібну інформацію.

Дані від клієнта передаються у шину даних (**9a**), що забезпечує потоковий інтерфейс для прийому та обробки інформації. Частина оброблених, агрегованих даних (**9b**) передається у хмарне сховище (**10**) для зберігання, інша частина – у локальне (**9c**).

Нарешті, сервер надає відповідь у вигляді HTML і відправляє її назад у браузер користувача, через балансер навантаження (**2**).

HTML-сторінка містить Javascript код та CSS, які завантажуються із CDN сховища (**11**).

Браузер відображає сторінку, яку бачить користувач.

Розглянемо більш детально кожен із компонентів.

1. **Система доменних імен (DNS)** – це ієрархічна та децентралізована система імен комп'ютерів, служб чи інших ресурсів, підключених до Інтернету чи приватної мережі.

Користувачі отримують доступ до інформації в Інтернеті через доменні імена, наприклад, google.com або wikipedia.org. DNS перетворює доменні імена на IP-адреси, щоб браузери могли завантажувати ресурси із Інтернету у зручному для користувачів вигляді.

Кожен пристрій, підключений до Інтернету, має унікальну IP-адресу, яку використовують інші машини для пошуку пристрою. DNS-сервери усувають потребу запам'ятовувати IP-адреси, такі як, наприклад, 192.168.1.1 (IPv4).

2. **Балансер навантаження** виконує функцію ефективного розподілу вхідного мережевого трафіку через групу серверів, також відомих як ферма серверів або пул серверів.

Сучасні веб-сайти з високим трафіком повинні обслуговувати сотні тисяч (якщо не мільйони) одночасних запитів користувачів або клієнтів і швидко та надійно повертати правильні текстові, зображення, відео чи дані програми. Для економічного масштабування і задоволення цих великих обсягів даних сучасна обчислювальна практика зазвичай вимагає додавання більше серверів.

Балансер навантаження виступає в ролі «дорожнього поліцейського», що знаходиться перед серверами і маршрутизує запити клієнтів на усі сервера, які здатні виконати ці запити, таким чином, щоб максимально використовувати швидкість, потужність і гарантувати, що жоден сервер не буде перевантажений, що може погіршити продуктивність. Якщо один сервер вийшов з ладу, балансер навантажень перенаправляє трафік на решту онлайн-серверів. Коли в групу серверів додається новий сервер, балансер навантаження автоматично починає надсилати на нього запити.

3. **Веб-сервери** виконують основну логіку бізнесу, яка обробляє запит користувача та пересилає HTML назад у браузер користувача. Веб-сервер зазвичай спілкуються із різноманітною інфраструктурою, що складається з даних, такі як бази даних, шари кешування, черги завдань, служби пошуку, мікросервіси, черги даних / журналів тощо.

Для обробки запитів користувачів, як правило, підключається принаймні два, або більше серверів.

4. **Бази даних** надають можливість визначення структури даних, вставки нових даних, пошуку даних, оновлення або видалення існуючих даних, проведення обчислень над даними тощо.

Крім того, у кожного мікросервісу, служби може бути власна база даних, яка ізольована від решти програми.

5. **Служба кешування** (*caching service*) надає сховище даних ключ-значення, що дозволяє зберігати та шукати інформацію із витратами часу близько *O(1)*. Зазвичай програми використовують кеш-сервіси для збереження результатів складних обчислень, щоб можна було отримати результати з кешу, а не перераховувати їх наступного разу, коли вони знадобляться. Додаток може кешувати результати із бази даних, відповіді від зовнішніх служб, HTML-сторінку для заданої URL-адреси тощо.

Дві найпоширеніші технології кешування сервера це – Redis і Memcache.

6. У **черзі задач** зберігається список завдань, які потрібно виконати асинхронно. Найпростішими є черги FIFO (*first-in-first-out*), хоча для більшості програм потрібна система черги з пріоритетом.

Кожен раз, коли додатку потрібна робота, яку потрібно запустити, або за певним регулярним графіком, або як це визначено діями користувача, сервер додає відповідну задачу до черги.

«Сервери роботи» обробляють завдання, опитуючи чергу, щоб визначити, чи є задача, яку потрібно виконати.

7. **Сервіс повнотекстового пошуку** використовує індексування даних для швидкого пошуку інформації по ключовим словам.

Хоча повнотекстовий пошук можна здійснювати безпосередньо за допомогою баз даних (наприклад, MySQL підтримує повнотекстовий пошук), типово запускати окрему «службу пошуку», яка обчислює та зберігає індексовані дані. Найпопулярнішою повнотекстовою пошуковою платформою сьогодні є Elasticsearch, хоча є й інші варіанти, такі як Sphinx або Apache Solr.

8. **Допоміжні сервіси (служби)** надають потрібну інформацію за запитом від сервера. Наприклад, служба облікових записів зберігає дані користувачів, що дозволяє отримувати усю необхідну інформацію про клієнтів додатку.

9. **Конвеєр даних** використовується, щоб забезпечити можливість збирання, обробки, фільтрування, зберігання та аналізу даних.

10. **Хмарне сховище даних** – це простий і масштабований спосіб зберігання, доступу та обміну даними через мережу Інтернет. Для доступу до даних використовується REST API.

Amazon S3 на сьогоднішній день є найпопулярнішим хмарним сховищем, яке доступне сьогодні.

11. **CDN** (*Content Delivery Network, Мережа Доставки Контенту*) – технологія, що забезпечує можливість доступу до даних, таких як статичний HTML, CSS, Javascript і зображення в Інтернеті, набагато швидше, ніж їх обслуговування з одного сервера-джерела.

CDN працює, поширюючи дані на багатьох серверах по всьому світу, так що користувачі в кінцевому рахунку завантажують дані із найближчого сервера, що дозволяє значно скоротити завантаження усієї сторінки.

1.3. Патерни побудови веб-додатків

1.3.1. Монолітний додаток

Монолітний додаток повністю замкнутий в контексті поведінки. Під час роботи додаток може взаємодіяти з іншими службами або сховищами даних, проте уся поведінка реалізується у власному, одиничному процесі, а додаток зазвичай розгортається як один елемент. Для горизонтального масштабування такий додаток зазвичай цілком дублюється на декількох серверах або віртуальних машинах.

Термін «моноліт» означає – «усе складене в одне ціле», у якому різні компоненти об'єднуються в єдину програму з однієї платформи.

Монолітний додаток представлений у вигляді однієї WAR-програми або Node-додаток з однією точкою входу тощо.

На рис. 1.6 зображено приклад монолітної архітектури: додаток складається із декількох сервісів, які взаємодіють із БД. Клієнти, використовуючи браузер або мобільні пристрої, взаємодіють із веб-сервером.

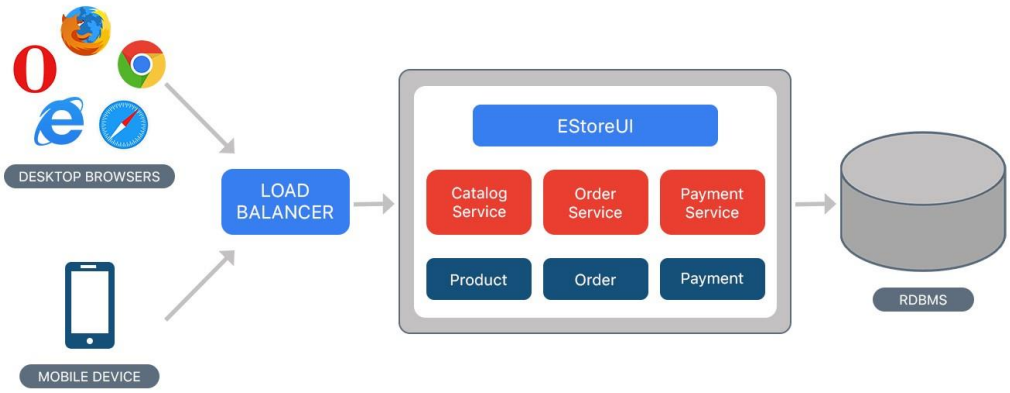


Рис.1.6. Приклад монолітної архітектури

Переваги:

- мала кількість ролей при розробці та експлуатації;
- простота у розгортанні – потрібно лише скопіювати «упаковану» програму на веб-сервер;
- легко забезпечувати транзакційну цілісність даних;
- простота у розробці – новий проект набагато простіше розробляти опираючись на монолітну архітектуру;
- легко масштабувати горизонтально, запустивши декілька копій додатку і балансер навантаження.

Недоліки:

- технічне обслуговування – якщо додаток занадто великий і складний, обслуговування може ускладнитись;
- єдиний технологічний стек технологій для усіх компонентів додатка;

- великі часові і людські витрати при модифікації і рефакторингу додатку із складною бізнес-логікою;
- перезапуск усього додатку при появі нової функціональності;
- великий «розмір» програми може сповільнити час запуску і розгортання;
- монолітні програми можуть бути складними для масштабування, коли модулі мають різні вимоги до апаратних ресурсів;
- ненадійність – помилка в будь-якому модулі (наприклад, витік пам'яті) може потенційно привести до збою усього додатку.

1.3.2. Мікросервісний підхід

Мікросервісна архітектура – це підхід до розробки додатків, у якому великий додаток складається із набору модульних служб, сервісів, мало пов’язаних модулів / компонентів. Кожен модуль реалізує конкретну бізнес-логіку та використовує простий, чітко визначений інтерфейс для спілкування з іншими сервісами (рис. 1.7).

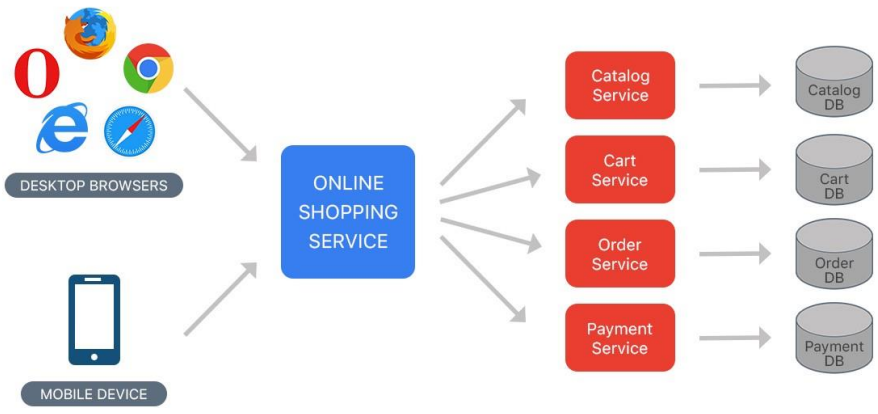


Рис. 1.7. Приклад мікросервісної архітектури

Замість обміну даними із єдиною базою даних, як у випадку моноліт-архітектури, кожен мікросервіс має свою базу даних. Єдина БД на сервіс забезпечує слабку зв’язність між компонентами додатку. Крім того, сервіс

може використовувати тип бази даних, який найкраще відповідає його потребам (наприклад, SQL або NoSql).

Переваги:

- можливість безперервно розробляти та розгортати великі, складні програми;
- гнучкість у масштабуванні;
- декомпозиція бізнес-логіки додатку;
- легко тестувати роботу одного мікросервіса (а не усього додатку у цілому);
- можливість підтримки декількох версій сервісу одночасно;
- розгортання – сервіси можна розгортати незалежно від інших сервісів;
- обмежена відповідальність – кожна команда несе відповідальність за один або декілька окремих сервісів. Кожна команда може розробляти, розгортати та масштабувати свої компоненти незалежно від усіх інших команд.
- швидкий запуск окремого мікросервіса;
- ізоляція відмов – проблема одного сервісу не впливає на роботу інших.

Недоліки:

- складність підтримки розподілених транзакцій;
- додаткова складність із розгортанням додатків та створенням розподіленої системи;
- необхідно реалізувати механізм міжсервісного зв'язку і стандартизувати протокол та формат повідомлень;
- необхідність забезпечувати сумісність для усіх існуючих сервісів;
- моніторинг та логування інформації декількох мікросервісів.

1.3.3. Безсерверна архітектура

Безсерверна архітектура (*serverless-architecture*) – це спосіб створення і запуску додатків і сервісів без необхідності управління інфраструктурою. Додаток, як і раніше працює на серверах, але управління цими серверами бере на себе сторонній сервіс. Це звільняє від необхідності займатися виділенням ресурсів, масштабуванням і обслуговуванням серверів для запуску додатків, баз даних і систем зберігання даних.

Безсерверні обчислення – це модель виконання комп'ютерного коду, в якій розробники звільняються від обслуговування інфраструктурними компонентами системи. Ця тенденція також відома як функція як послуга (*FaaS*), коли постачальник хмарних послуг відповідає за запуск і зупинку контейнерної платформи функції, перевірку безпеки інфраструктури, скорочення витрат на обслуговування, поліпшення масштабованості.

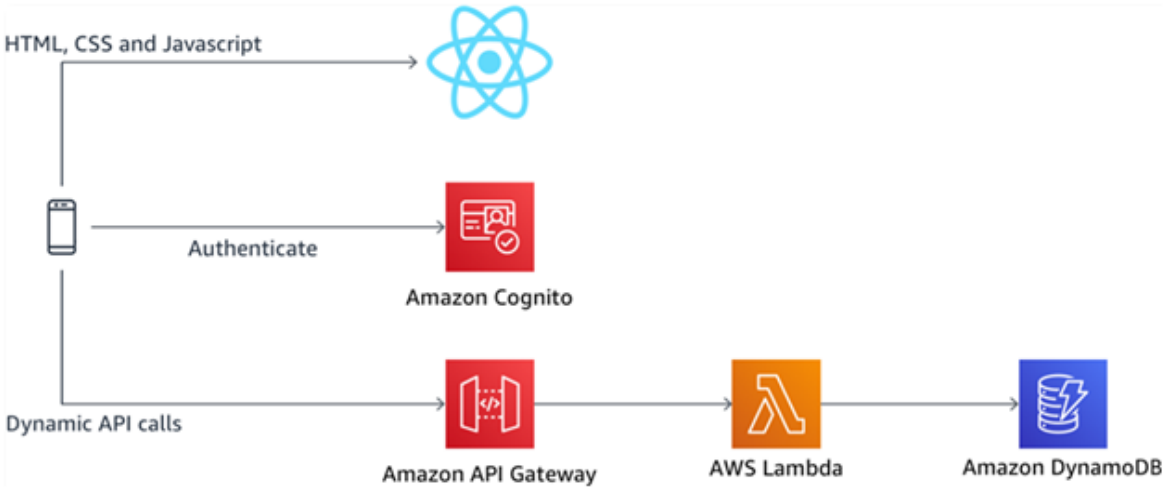


Рис. 1.8. Приклад безсерверної архітектури

У безсерверних додатках спеціальні програмні компоненти активізуються тільки тоді, коли надходить запит. Після обробки запиту компонент «засинає». Дані компоненти часто розташовуються у керованому середовищі, яке відповідає за масштабованість і стежить за життєвим циклом.

Переваги:

- економія часу і накладних витрат;
- масштабування відповідно до навантаження;
- немає необхідності займатись інфраструктурними питаннями.

Недоліки:

- складність моніторингу і виявлення помилок;
- неможливість внесення змін у інфраструктури без узгодження із постачальником послуг;
- можливі затримки у роботі сервісу.

1.3.4. Порівняльна характеристика підходів

Опираючись на наведені вище особливості різних підходів до проектування додатків, створимо порівняльну характеристику за визначеними критеріями.

Теорема CAP (*теорема Брюєра*) – евристичне твердження про те, що в будь-який реалізації розподілених обчислень можливо забезпечити не більше двох з трьох наступних властивостей (табл. 1.1):

- узгодженість даних (*consistency*) – у всіх вузлах розподіленої системи в один момент часу усі дані однакові і не суперечать один одному;
- доступність (*availability*) – на будь-який запит до будь-якого вузла розподіленої системи приходить очікувана відповідь, але немає гарантії, що всі відповіді в один момент часу будуть збігатися;
- стійкість до розподілу (*partition tolerance*) – поділ вузлів розподіленої системи на кілька ізольованих сегментів не призводить до неочікуваної роботі системи.

Таблиця 1.1

Характеристика CAP

Архітектура	Узгодженість	Доступність	Стійкість до розподілу
Моноліт	+	−	−
Мікросервіс	−	+	+
Безсерверна	−	+	+

Вертикальне масштабування (*scaling up*) – збільшення кількості доступних для ПО ресурсів за рахунок збільшення потужності з серверів. У даному випадку перевагу віддають потужнішому апаратному забезпеченню.

Горизонтальне масштабування (*scaling out*) – збільшення кількості серверів, об'єднаних в кластер серверів або процесів за рахунок ОС.

У табл. 1.2 показано допустимість типу масштабування у залежності від виду архітектури.

Таблиця 1.2

Характеристика по типу масштабування

Архітектура	Вертикальне	Горизонтальне
Моноліт	+	−
Мікросервіс	+	+
Безсерверна	+	+

У табл. 1.3 вказана приблизна оцінка вимог апаратних ресурсів інфраструктури до архітектурного рішення.

Позначення: S – small (*невисокі вимоги*), M – middle (*середні вимоги*), L – large (*значні вимоги*).

Типи ресурсів:

- Пам'ять (*RAM*);
- Ресурси процесора (*CPU*);
- Місце на диску (*HDD / SSD*);
- Комунікація (*LAN*);

Таблиця 1.3

Характеристика по типам ресурсів

Архітектура	RAM	CPU	HDD/SSD	LAN
Моноліт	**M**	**M**	**M**	**S**
Мікросервіс	**L**	**L**	**S**	**L**
Безсерверна	**L**	**L**	**S**	**L**

Висновок

Високі темпи змін і висока складність можуть бути факторами, які змушують вибрати архітектуру мікросервіса. Можливість безперервно розробляти та розгортати великі, складні програми буде тільки сприяти активному росту бізнесу.

Навпаки, коли ще немає чіткої предметної області, почати з моноліту буде правильним рішенням, проте варто робити служби модульними. Це полегшить завдання, якщо буде вирішено розділити моноліт на кілька мікросервісів.

Завдяки безсерверній архітектурі, немає необхідності займатися виділенням ресурсів, масштабуванням і обслуговуванням серверів для запуску додатків, а отже можна повністю зосередитись на вимоги бізнесу і розробці додатку.

РОЗДІЛ 2
АНАЛІЗ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1. Характеристика і принципи побудови мікросервісів

Мікросервіси, або архітектура мікросервісів – це архітектурний стиль, який декомпозує додаток як сукупність невеликих автономних сервісів, модельованих навколо бізнес-домену або бізнес-логіки додатку.

Мікросервісна архітектура стала одним із підходів до розробки прогресивних додатків сьогодні.

Принципи на яких побудована мікросервісна архітектура:

- Масштабованість
- Доступність
- Стійкість до відмов
- Незалежність, автономність
- Децентралізоване управління
- Ізоляція відмов
- Автоматичне виявлення сервісів
- Безперервна доставка

Мета мікросервісів – збільшити швидкість випуску додатків, розклавши додаток на невеликі автономні сервіси, які можна розгорнути самостійно. Архітектура мікросервісів також приносить певні проблеми. Розглянемо основі архітектурні проблеми і методи їх вирішення.

Мікросервісний підхід має декілька моделей дизайну архітектури, їх можна розділити на п’ять моделей (рис. 2.1).

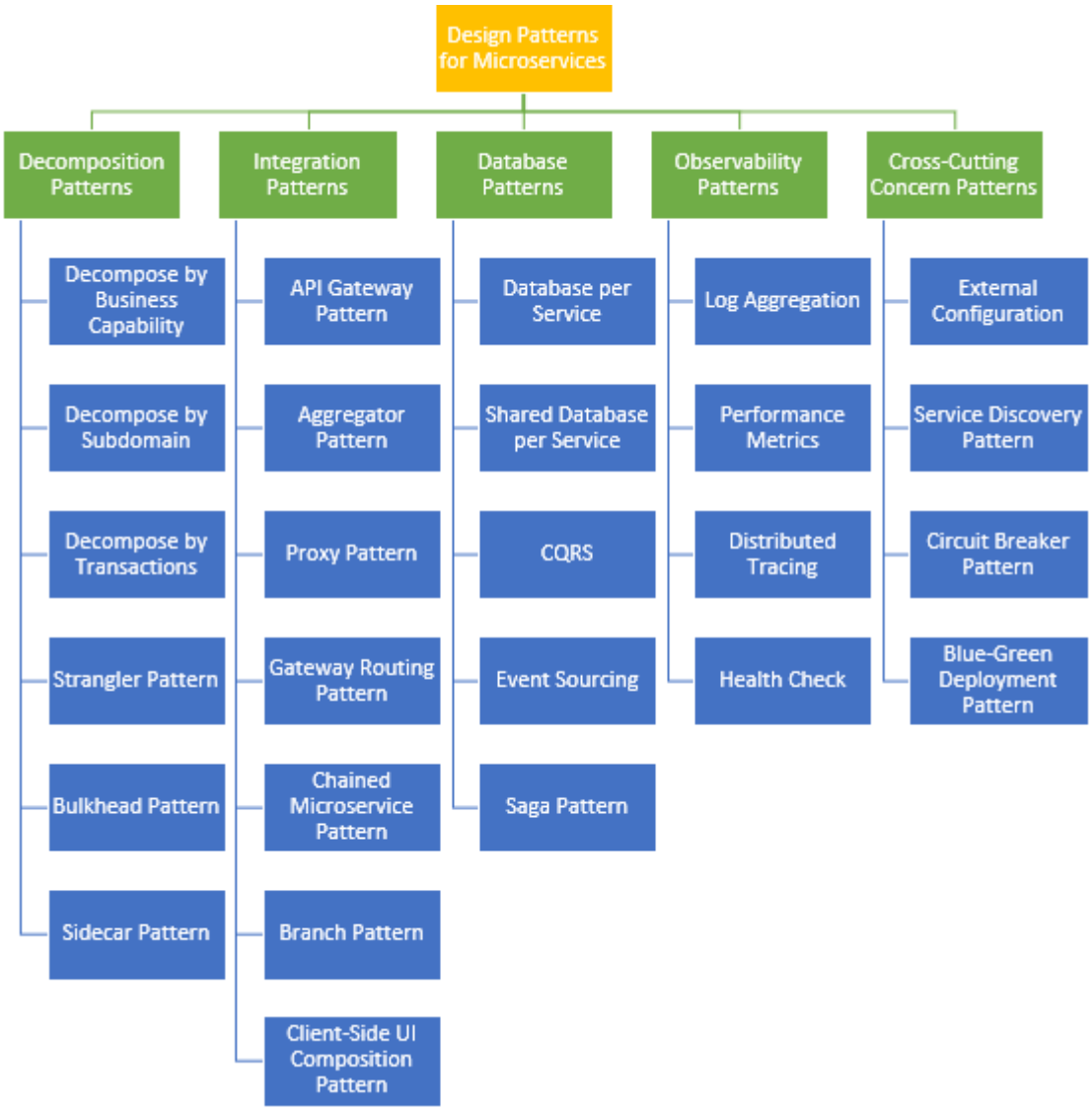


Рис. 2.1. Патерни мікросервісної архітектури

2.1.1. Декомпозиція вимог бізнесу

Бізнес-вимога (*Business Domain, бізнес-можливість*) – те що виконує організація, а це у свою чергу сприяє досягненню її бізнес-цілей. Наприклад, обробка кошика замовлень в інтернет-магазині – це бізнес-вимога, яка допомагає досягти більш широкої бізнес-цілі, що складається в забезпеченні можливості покупки товарів користувачами через інтернет. У кожного комерційного підприємства є безліч бізнес-вимог, які разом утворюють його загальну бізнес-функцію.

Таким чином мікросервіс реалізує бізнес-можливість в цілому і повністю автоматизує її виконання. Проте можливо, що мікросервіс реалізує тільки

частину бізнес-можливості і, таким чином, автоматизує її виконання лише частково. В обох випадках область дії мікросервіса дорівнює бізнес-можливості.

Декомпозуючи систему таким чином, застосовують принцип єдиної відповідальності. Система розподіляється на основі бізнес-вимог та визначаються послуги, що задовольняють вимоги бізнесу (рис. 2.2).

Набір вимог для даного бізнесу залежить від типу бізнесу. Наприклад, робота страхової компанії зазвичай включає у себе продажі, маркетинг, обробка пропозицій і претензій, виставлення рахунків, відповідність. Отже система, декомпозується на сервіси: сервіс продажу, сервіс маркетингу, сервіс рахунків і тд.

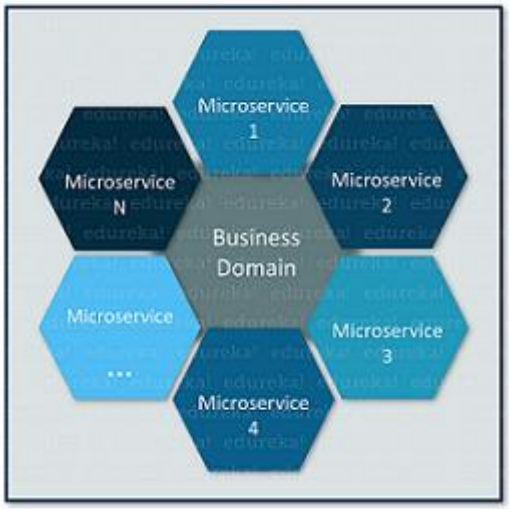


Рис. 2.2. Декомпозиція на основі бізнес-вимог

2.1.2. Декомпозиція субдомену

Предметно-орієнтоване проектування (*domain-driven design, DDD*) – підхід до проектування програмного забезпечення, заснований на моделюванні предметної області.

Обмеженим контекстом (*bounded context*) в предметно-орієнтованому проектуванні називається деяка частина предметної області, в межах якої терміни, поняття зберігають своє значення. Обмежений контекст визначає

частину предметної області, всередині якої контекст є однаковим. В межах одного обмеженого контексту підприємству може знадобитися виконувати кілька дій, кожна з яких, ймовірно, є бізнес-можливість.

Декомпозиція програми з використанням бізнес-можливостей може стати хорошим початком, але можна зіткнутись із так званими «класами богів» (*God Classes*), розділити які буде непросто. Дизайн, на основі DDD, визначає цей проблемний простір програми як бізнес-домен. Домен складається з декількох субдоменів (*піддоменів*), і кожен субдомен відповідає різній частині бізнесу.

Наприклад, бізнес-можливість включає у себе сервіс замовлень – бізнес-домен. Даний сервіс можна декомпозувати на декілька піддоменів:

- сервіс каталогу товарів;
- послуги з керування складом;
- послуги з управління замовленнями;
- послуги з управління доставкою.

Отже, декомпозиція субдомену використовується для дроблення бізнес-домену задля збереження автономності та атомарності кожного з модулів.

2.1.3. Шаблон Strangler – міграція на мікросервіси

Шаблон Strangler (Ш*аблон придушення*) – покрокова міграція застарілої системи з поступовою заміною певних компонентів новими додатками і службами. Компоненти застарілої системи поступово видаляються, і з часом нова система повністю візьме на себе всі її функції, що дозволяє вивести стару систему з експлуатації.

Засоби розробки, технології розміщення і архітектури, на основі яких створюється будь-яка система, часто застарівають з плином часу. У міру додавання нових функцій і можливостей неухильно зростає складність старих

додатків, що ще більше ускладнює обслуговування і додавання нових компонентів.

Але повна заміна складної системи часто пов'язана з величезними труднощами. У багатьох випадках краще переходити на нову систему поступово, зберігаючи стару систему для обробки тих можливостей, які ще не реалізовані в новій (рис. 2.3). Виконання двох різних версій однієї програми означає, що клієнти повинні знати, в який з них знаходяться потрібні компоненти.

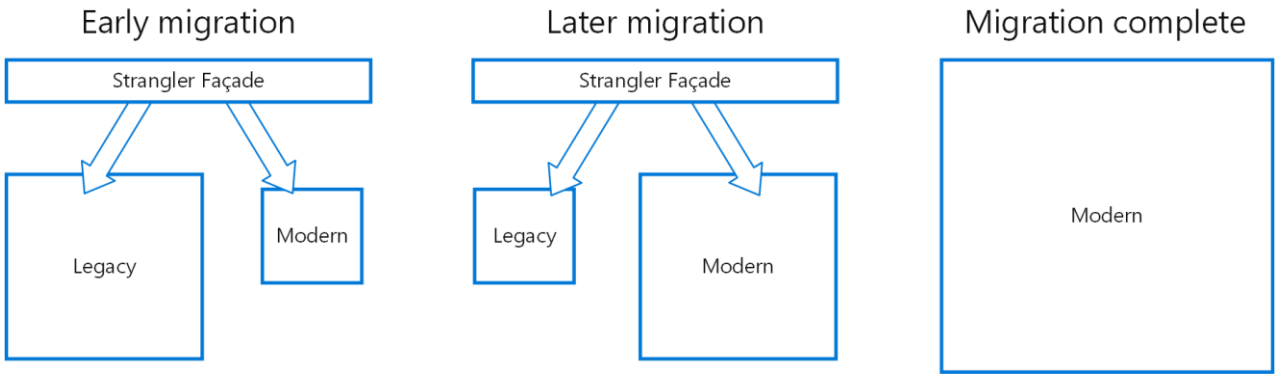


Рис. 2.3. Поетапний перехід на новий компонент

Згідно із шаблоном, потрібно поетапно замінювати компоненти новими додатками і службами. Створюється оболонка (фасад), яка перехоплює всі запити до застарілих систем серверної частини. Дана оболонка буде вибирати, куди направити такі запити: до застарілого компоненту додатку або до нової служби. Так можна перенести компоненти в нову систему поступово, не змінюючи інтерфейс для користувачів, які навіть не помітять, що відбувається поетапна міграція.

Дана модель дозволяє знизити ризики міграції та розподілити зусилля розробників по часу. Поки оболонка безпечно направляє користувачів до правильного додатку, можна додавати компоненти в нову систему в комфортному темпі, зберігаючи при цьому працездатність старого додатка. Через деякий час всі функції будуть перенесені в нову систему і застаріла

система більше не буде потрібна. Коли цей процес завершиться, можна відключити і видалити стару систему.

Даний шаблон використовується у випадках переходу із монолітної системи до мікросервісної.

2.1.4. Шаблон Bulkhead – ізоляція відмов

Шаблон Bulkhead (*Шаблон обкладинки*) – це тип архітектури програми, яка стійка до відмов. Згідно із даною архітектурою, елементи програми виділяються в групи, так що, якщо один не працює, інші продовжуватимуть функціонувати.

Рішенням є розділення екземплярів служб на декілька груп відповідно до характеру навантаження від користувачів і вимогами доступності. Такий підхід дозволяє ізолювати відмови (збої), зберігаючи працездатність служб хоча б для деяких користувачів навіть під час нештатної ситуації.

Аналогічно можна розділити ресурси для споживача, щоб виклики до однієї службі не впливали на доступність ресурсів для виклику інших служб.

Така схема надає наступні переваги:

- ізоляція споживачів і служб від каскадних відмов. Проблема, яка зачіпає споживача або службу, буде ізольована в одній конкретній групі, що не буде впливати на інші модулі;
- збереження функціональності навіть в разі збою служби. Інші служби та функції програми продовжать працювати;
- різна пріоритетність ресурсів для груп.

На рис. 2.4 показано клієнтів, які викликають один сервіс. Кожному клієнту призначений окремий екземпляр цієї служби. Клієнт №1 взаємодіє із пошкодженим сервісом. Так як кожен екземпляр служб ізольований від інших, інші клієнти спокійно продовжують працювати з цією службою.

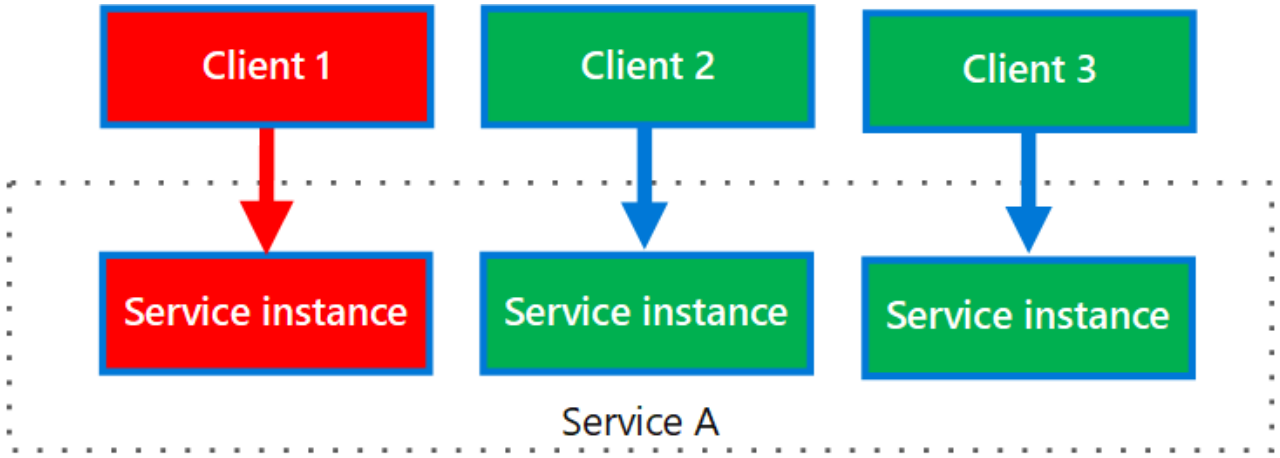


Рис. 2.4. Ізоляція відмов

2.1.5. Шаблон Sidecar – автономність сервісу

Шаблон Sidecar (*Шаблон розширення*) – для забезпечення ізоляції і інкапсуляції, компоненти програми розгортаються в окремому процесі або контейнері. Даний шаблон також може включати можливість створення додатків, що складаються з різнорідних компонентів і технологій.

Для додатків і служб часто потрібні пов'язані функції, такі як моніторинг, ведення журналів, конфігурація і мережеві служби. Ці периферійні завдання можна реалізувати в якості окремих компонентів або служб.

Якщо вони тісно інтегровані в додатку, то можуть виконуватися в одному процесі в якості додатку, за рахунок чого забезпечується ефективне використання загальних ресурсів. Однак це також означає, що вони неефективно ізольовані і збій в одному з цих компонентів може вплинути на інші або ж на кожну з програм. В результаті компонент і «батьківський» додаток тісно пов'язані один з одним.

Якщо додаток ділиться на служби, кожну з них можна створити з використанням різних мов і технологій. При цьому забезпечується додаткова гнучкість, але у кожного компонента є власні залежності, такі як доступ до базової платформі і ресурсів батьківського додатка. Крім того, розгортання цих функцій в якості окремих служб може призвести до затримок у додатку.

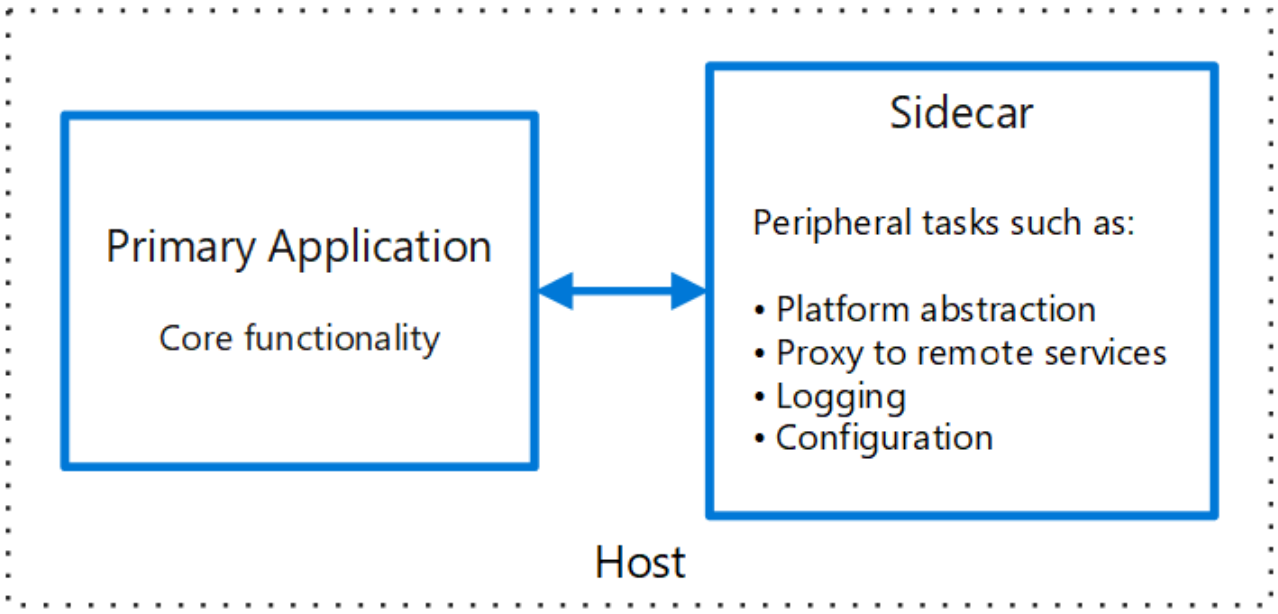


Рис. 2.5. Розділення служб по різним процесам

Згідно із шаблоном, рішенням є розділення сервісу по окремим, власним процесам або контейнерам, надаючи однорідний інтерфейс для служб на платформі для різних мов (рис. 2.5).

2.2. Міжпроцесорна взаємодія

Додаток на основі мікросервісів являє собою розподілену систему, що працює на кількох процесах або службах, іноді навіть не декількох серверах або вузлах. Зазвичай кожен екземпляр служби – це процес.

Таким чином, служби повинні взаємодіяти по протоколу внутрішьопроцесорної взаємодії, наприклад HTTP, AMQP або бінарного протоколу, такому як TCP, в залежності від характеру кожної служби.

Зазвичай використовуються два протоколи – запити і відповіді HTTP з вихідними API (в основному для запитів) і легкі асинхронні повідомлення при передачі оновлень у декілька мікросервісів.

Клієнт і служби можуть взаємодіяти через різні типи зв'язку в залежності від сценарію і цілей. Ці типи зв'язку можна розділити на два напрямки.

Перша група визначає, протокол синхронний чи асинхронний (рис. 2.6):

- синхронний протокол. HTTP – це синхронний протокол. Клієнт відправляє запит і чекає відповіді від служби. Тут важливо, що протокол (HTTP / HTTPS) є синхронним і код клієнта зможе продовжити виконання завдання тільки після отримання відповіді від HTTP-сервера;
- асинхронний протокол. Інші протоколи, наприклад AMQP (протокол, підтримуваний багатьма операційними системами і хмарними середовищами), використовують асинхронні повідомлення. Код клієнта або відправник повідомлення зазвичай не очікує відповіді. Він просто відправляє повідомлення, як при відправці повідомлення в чергу RabbitMQ або іншого брокера повідомлень.

Друга група визначає, має запит одного або кількох одержувачів:

- один одержувач – кожен запит повинен оброблятися тільки одним одержувачем або службою;
- декілька одержувачів – кожен запит може оброблятися різною кількістю одержувачів – від жодного до декількох. Такий тип взаємодії повинен бути асинхронним. Наприклад, механізм «publish-subscribe», який використовується в таких шаблонах, як архітектура, керована подіями. Він заснований на інтерфейсі шини подій або брокера повідомлень, коли події оновлюють дані в декількох мікросервісах. Зазвичай це реалізується через службову шину або подібний об'єкт, наприклад службову шину Azure, за допомогою тем і підписок.

Додаток на базі мікросервісів часто використовує комбінацію цих стилів взаємодії. Найбільш поширений тип – взаємодія з одним одержувачем з синхронного протоколу, наприклад HTTP або HTTPS, при виклику звичайної служби веб-API HTTP. Для асинхронного взаємодії між мікросервісами зазвичай використовуються протоколи повідомлень.

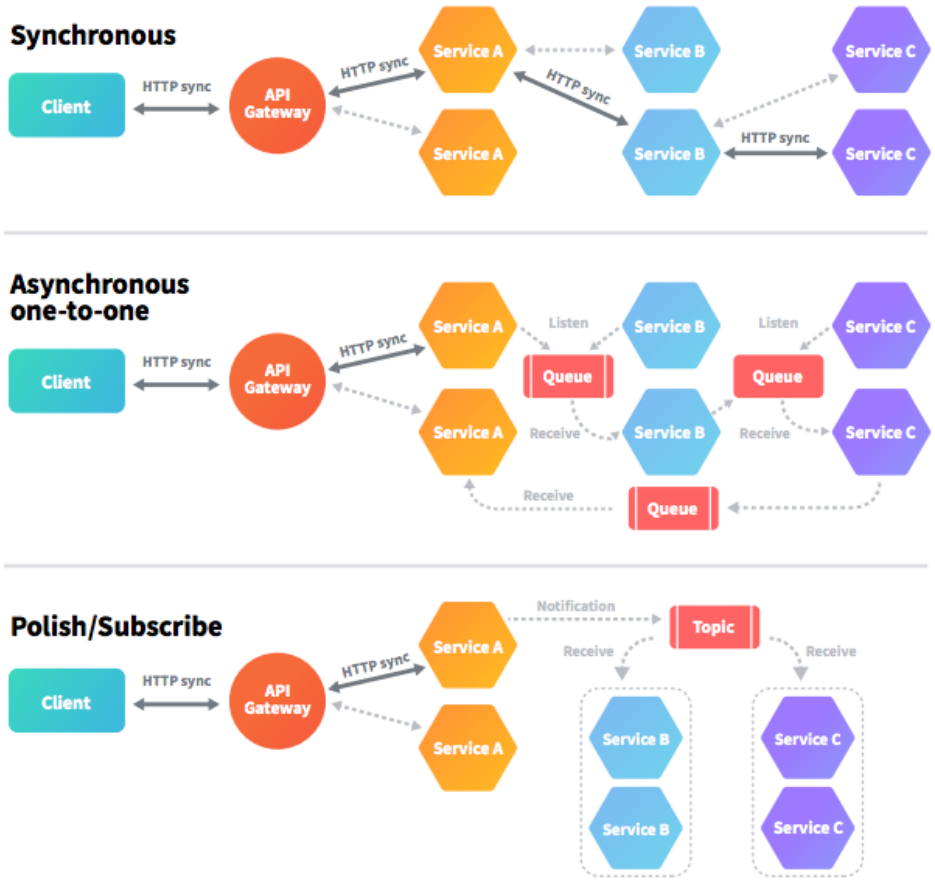


Рис. 2.6. Типи взаємодій мікросервісів

2.2.1. Виявлення сервісів

Головна ідея, що лежить в основі виявлення сервісів (*service discovery*), полягає в тому, що будь-який новий екземпляр додатку повинен бути у змозі програмно визначити деталі свого поточного оточення. Це необхідно для того, щоб новий екземпляр міг підключитися до існуючого оточення, тобто зв’язатись із іншими сервісами без ручного втручання. Інструменти пошуку послуг зазвичай реалізовані у вигляді глобально доступних реєстрів, які зберігають інформацію про екземпляри додатків і сервісів, запущених у даний момент.

Хоча основна мета інструментів з пошуку послуг складається в наданні деталей підключення для зв'язку компонентів разом, а саме *ip* та *порт*, вони так само можуть використовуватися для зберігання будь-якого типу конфігураційної інформації. Багато додатків використовують цю можливість у своїх цілях, зберігаючи свою конфігураційну інформацію за допомогою

інструменту пошуку послуг. Наприклад, конфігураційною інформацією сервіса може бути шлях на диску для логування інформації, який може залежати від того, на якому оточені запущений додаток.

Виявлення відмов може бути реалізовано кількома способами. Суть полягає в тому, що в разі відмови компонента система пошуку послуг повинна дізнатися про це і відображати факт, що зазначений компонент більше не доступний. Даний тип інформації необхідний для мінімізації відмов додатку або сервісу.

На рис. 2.7 зображений реєстр сервісів який виконує функції виявлення сервісів. Запит від клієнта проходить через балансер навантаження, який зв'язується із реєстром сервісів, щоб отримати розташування (ip та порт) усіх екземплярів додатку, після, направляє запит до потрібного екземпляру.

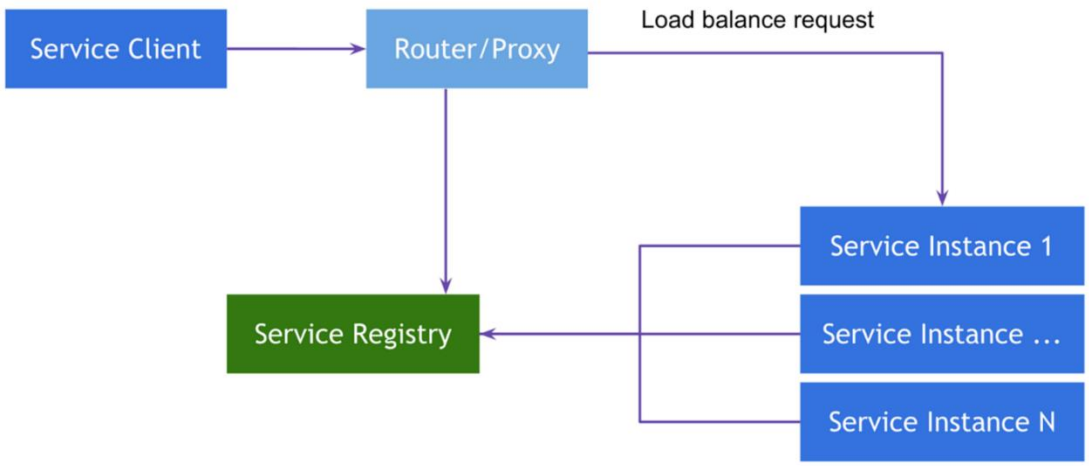


Рис. 2.7. Реєстр сервісів

Системи виявлення сервісів дозволяють задавати значення з налаштованим тайм-аутом. Компонент може встановити значення з тайм-аутом і регулярно пінгувати (*ping*) систему виявлення сервісів. Якщо відбувається відмова компонента, і досягається тайм-аут, інформація про підключення для цього компонента видаляється зі сховища. Тривалість тайм-ауту, в основному, залежить від того, як швидко додаток повинен реагувати на відмову компонента.

2.2.2. Синхронна взаємодія

Синхронна взаємодія – це стиль спілкування, коли один сервіс чекає, поки з’явиться відповідь від іншого. Його концептуальна простота дозволяє просту реалізацію, що робить її добре придатною для більшості ситуацій.

Синхронний зв'язок тісно пов'язаний з протоколом HTTP. Однак інші протоколи залишаються не менш розумним способом реалізації синхронного зв'язку, хороший приклад альтернативи – запити RPC.

Як механізм взаємодії за допомогою синхронних запитів і відповідей зазвичай використовуються HTTP і REST, особливо якщо спілкування служби ведеться за межами вузла Docker або кластера.

REST (*Representational State Transfer, Передача Стану Представлення*) – архітектурний стиль взаємодії компонентів розподіленого додатка в мережі. REST є узгоджений набір обмежень, що враховуються при проектуванні розподіленої системи. У певних випадках (інтернет-магазини, пошукові системи, інші системи, засновані на даних) це призводить до підвищення продуктивності і спрощення архітектури. У широкому сенсі компоненти в REST взаємодіють на зразок взаємодії клієнтів і серверів у Всесвітній павутині.

REST підхід заснований на HTTP-протоколі і тісно пов'язаний з ним. Він приймає HTTP-команди, наприклад GET, POST і PUT. REST – це найпоширеніший архітектурний підхід до взаємодії при створенні служб.

Якщо взаємодія між службами здійснюється в межах вузла Docker або кластера мікросервісів, можна використовувати двійковий формат взаємодії (наприклад, WCF за допомогою TCP і двійкового формату).

Коли клієнт використовує взаємодію типу «запит-відповідь», він передбачає, що відповідь прийде швидко, менше ніж через секунду або максимум через кілька секунд. Якщо відповідь затримується, необхідно реалізувати асинхронну взаємодію на основі шаблонів обміну повідомленнями та технологій обміну повідомленнями.

2.2.3. Асинхронна взаємодія

Взаємодія на основі подій є асинхронною. Це означає, що сервіс, який публікує подію у шину повідомлень (*message bus*), виконує звернення до підписаних на ці події сервіси. Сервіси-підписники, в міру своєї готовності до обробки, опитують брокера повідомлень на наявність нових подій.

Хоча самі опитування полягають у виконанні синхронних запитів, взаємодія залишається асинхронною, оскільки публікація подій не залежить від виконання підписниками опитувань на предмет подій.

Основний принцип шини повідомлень – концентрація обміну повідомленнями між різними системами через єдину точку, в якій, при необхідності, забезпечується транзакційний контроль, перетворення даних, збереження повідомлень. Всі налаштування обробки і передачі повідомлень передбачаються також сконцентрованими в єдиній точці, таким чином, що при заміні будь-якої системи, підключеної до шини, немає необхідності в переналаштуванні інших систем.

Події використовуються, коли мікросервісу необхідно реагувати на щось, що відбувається в іншому мікросервісі.

Як правило, виділяються наступні ключові можливості:

- підтримка синхронного і асинхронного способу виклику служб;
- використання захищеного транспорту, з гарантованою доставкою повідомлень, що підтримує транзакціону модель;
- статична і алгоритмічна маршрутизація повідомлень;
- доступ до даних з сторонніх інформаційних систем за допомогою готових або спеціально розроблених адаптерів;
- обробка та перетворення повідомлень;
- оркестровка і хореографія служб;
- різноманітні механізми контролю і управління (аудити, протоколювання).

Конкретні програмні продукти зазвичай також містять готові адаптери для з'єднання з конкретним прикладним програмним забезпеченням, а також можуть включати API для створення таких адаптерів.

RabbitMQ – це додаток для роботи з асинхронними чергами повідомлень (*message-queueing*), ще його називають брокер повідомлень (*message broker*) або менеджер черг (*queue manager*). Тобто, це програмне забезпечення в якому можуть бути створені черги до яких можуть підключатись різні додатки і передавати та отримувати повідомлення.

RabbitMQ може виступати як прошарок між декількома сервісами. Може використовуватися для зменшення навантаження і прискорення відгуку веб-додатків, оскільки завдання які зазвичай займають досить багато часу, можуть бути делеговані асинхронно третій стороні.

На рис. 2.8 зображена схема взаємодії видавця (*producer*) і підписника (*consumer*). Видавець передає подію брокеру повідомлень, який публікує її у чергу повідомлень. Таким чином, підписники, які підписались на деяку чергу, отримають своє повідомлення.

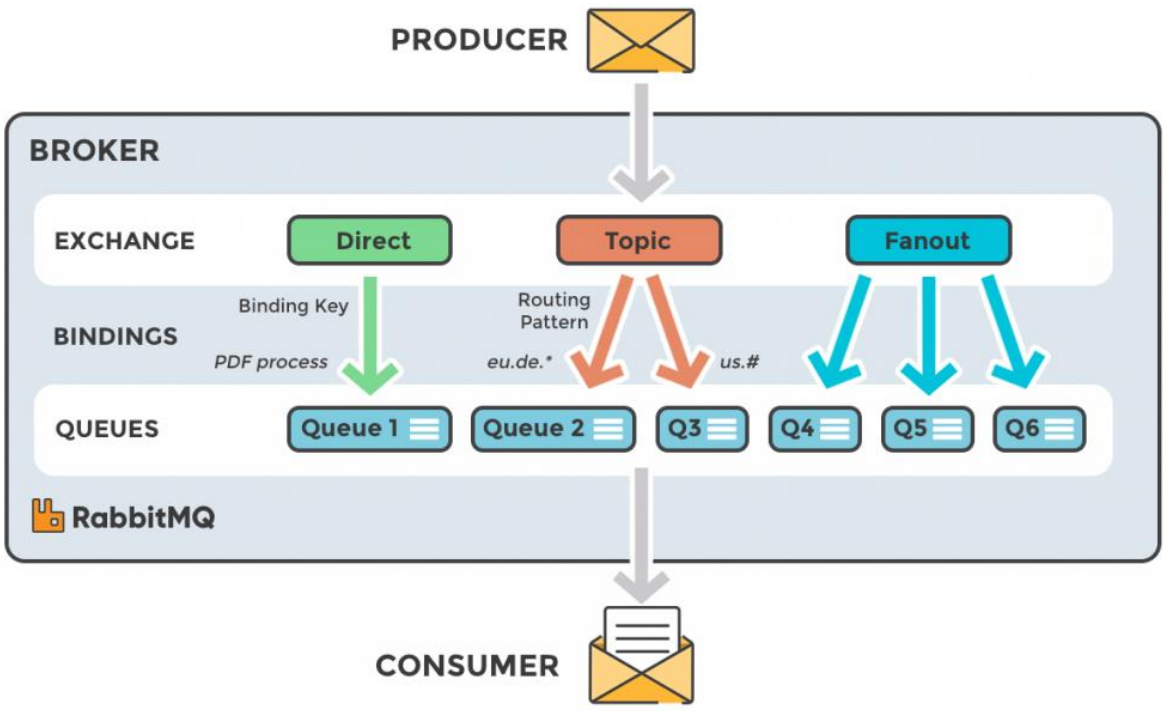


Рис. 2.8. Принцип взаємодії RabbitMQ

2.2.4. Формат повідомлень

В процесі розробки додатків з мікросервісною архітектурою неминуче виникає проблема вибору формату обміну даними між модулями. Вирішення цієї проблеми може мати значний вплив як на роботу програми, так і на трудомісткість подальшої модернізації. Час відгуку, обсяг переданої інформації по каналу зв'язку, розширюваність і портованість, необхідні ресурси і інші параметри можуть залежати від формату обміну даними.

Найчастіше в співтоваристві розробників, перевагу віддають одному з трьох найбільш використовуваних форматів обміну даними: XML, JSON, YAML.

XML (*Extensible Markup Language*) – простий, дуже гнучкий текстовий формат, який є підмножиною SGML (ISO 8879), який дозволяє визначати власні теги і атрибути. Мова називається розширюваною, оскільки не фіксується розмітка, яка використовується в документах: розробник вільний створити її відповідно до особливостей конкретної предметної області, будучи обмеженим лише синтаксичними правилами мови. Можливість створення власних тегів робить XML універсальним.

JSON (*Java Script Object Notation*) – являє собою полегшений формат обміну даними між комп'ютерами. JSON більш компактний, ніж XML, його конструкції легше аналізуються засобами JavaScript, для якого JSON є внутрішнім використовуваним типом даних. Основна сфера застосування JSON – програмування веб-додатків, де він служить альтернативою XML.

YAML - людиночитаємий формат серіалізації даних, концептуально близький до мов розмітки, але орієнтований на зручність введення-виведення типових структур даних багатьох мов програмування. В даний час акронім YAML інтерпретується як «YAML Is not Markup Language» («YAML – не мова розмітки»). У назві відображено історію розвитку: на ранніх етапах акронім був абревіатуру виразу «Yet Another Markup Language» («Ще одна мова розмітки») і навіть розглядався як конкурент XML, але пізніше був перейменований, щоб акцентувати увагу на даних, а не на розмітці документів.

У табл. 2.1 зображена порівняльна оцінка форматів даних [11].

Таблиця 2.1

Порівняльна оцінка форматів даних за шкалою зручності 1-5

Критерій	XML	JSON	YAML
Зручність читання	4	5	4
Простота серіалізації	5	5	5
Простота десеріалізації	5	5	5
Простота перевірки вхідних даних	4	4	3
Ефективність стиснення даних	3	5	1

Отже, аналізуючи характеристики кожного із представлених форматів даних, можна зробити висновок, що формат даних JSON, є найбільш оптимальним для використання у міжпроцесерній взаємодії мікросервісів.

2.3. Розгортання додатків

2.3.1. Безперервна інтеграція

Безперервна інтеграція (*Continuous Integration, CI*) – практика розробки програмного забезпечення, яка полягає в постійному злитті робочих копій додатку в загальну основну гілку розробки і виконанні частих автоматизованих збірок проекту для якнайшвидшого виявлення потенційних дефектів і рішення інтеграційних проблем.

Для застосування практики необхідно виконання ряду базових вимог до проекту розробки. Зокрема, вихідний код і все, що необхідно для побудови та тестування проекту, має зберігатися в репозиторії системи управління

версіями, а операції копіювання зі сховищ, збірки і тестування всього проекту повинні бути автоматизовані та легко викликатися із зовнішніх програм.

Для організації процесу безперервної інтеграції на виділеному сервері запускається служба, в завдання якої входять:

- отримання вихідного коду зі сховищ;
- збірка проекту;
- виконання тестів;
- розгортання готового проекту;
- відправка звітів.

На рис. 2.9 зображена схема безперервної інтеграції: спочатку розробник фіксує свої зміни у систему контролю версій (*git*), завантажує зміни у репозиторій контролю версій (*GitHub*), після чого система CI проводить модульне тестування коду (*unit test*), робить синтаксичний аналіз коду (*code analysis*), вимірює відсоток покриття тестами (*test coverage*).

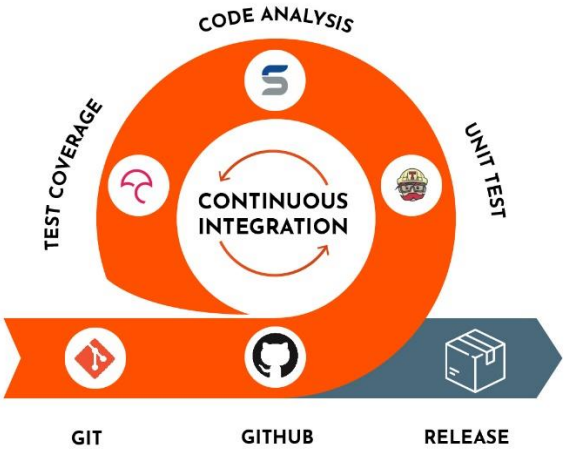


Рис. 2.9. Схема безперервної інтеграції

Збірка за розкладом, як правило, проводяться в неробочий час, вночі, плануються таким чином, щоб до початку чергового робочого дня були готові результати тестування. Для відмінності додатково вводиться система

нумерації збірок – зазвичай, кожна збірка нумерується натуральним числом, яке збільшується з кожною новою збіркою

Вихідні тексти та інші вихідні дані при взятті їх зі сховищ (сховища) системи контролю версій позначаються номером збірки. Завдяки цьому, точно така ж збірка може бути точно відтворена в майбутньому – досить взяти вихідні дані по потрібної мітці і запустити процес знову. Це дає можливість повторно випускати навіть дуже старі версії програми з невеликими виправленнями.

Одними із популярних системи CI є CircleCI, TeamCity, Jenkins.

2.3.2. Безперервна доставка

Безперервна доставка (*Continuous Delivery, CD*) – це підхід до розробки програмного забезпечення, при якому всі зміни, включаючи нові функції, зміни конфігурації, виправлення помилок і експерименти – поставляються користувачам максимально швидко і безпечно.

Увесь процес можна описати наступним чином:

- готовий зібраний пакет, після успішної інтеграції (CI), викладається на тестовий сервер;
- зацікавлені особи, отримують повідомлення про викладенні нової версії ПЗ на тестову платформу. Починається друга фаза тестування, запускаються інтеграційні, ручні, приймальні, UI тести і тд;
- після успішного проходження попередніх кроків, ми маємо готовий до публікації пакет, нової версії ПЗ.

Після підтвердження готовності пакета до релізу, приймається рішення про дату публікації, засноване на бізнес вимогах. Дуже важливо відзначити, що протягом усього процесу безперервної доставки, постійно отримується зворотний зв'язок.

Безперервна поставка дозволяє знизити ризики релізів, роблячи розгортання програмного забезпечення безболісним, безпечною подією, яка може бути виконана у будь-який час.

Автоматизуючи більшість операцій, таких як розгортання, налаштування оточення, тестування, ми скорочуємо час поставки нової функціональності до клієнтів.

Серед популярних рішень для CD є Maven, Crucible, Ant.

На рис. 2.10 зображена схема безперервної інтеграції. Етапи розробки (*Develop*) та збірки (*Build*) уже проведені на стадії CI. На етапі CD проводяться автоматизовані тести (*Automated Test*), та розгортання додатку на основному сервері (*Automated Deploy*), після чого вирішується чи запускати даний розгорнутий додаток у роботу (*Automated/Controller Release*).

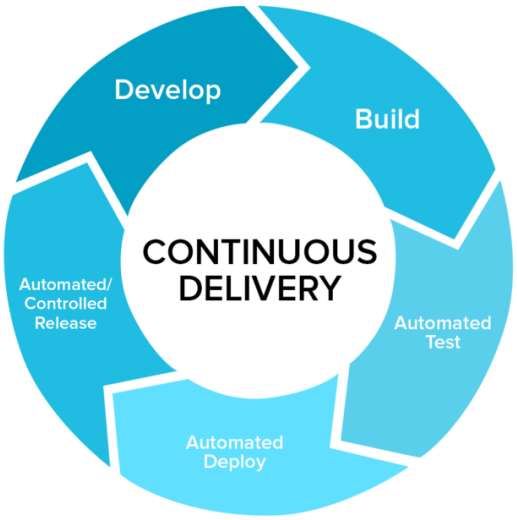


Рис. 2.10. Схема безперервної доставки

2.3.3. Стратегії розгортання

Кожен із мікросервісів представляє собою міні-додаток із своїми специфічними вимогами до розширення, ресурсу, масштабованості та моніторингу. Наприклад, потрібно запустити конкретну кількість екземплярів кожного сервісу в залежності від навантаження на цей сервіс. Крім того,

кожному екземпляру потрібно забезпечити відповідні ресурси процесора, пам'яті та I/O.

Розгортання мікросервісу має бути швидким, надійним та економічно ефективним.

2.3.3.1. Множина екземплярів на одному сервері

При використанні цього шаблону виділяється один або декілька фізичних або віртуальних серверів і запускається декілька екземплярів сервісів на кожному. У цілому це традиційний підхід до розгортання додатків.

Однією з основних переваг є ефективне використання ресурсів. Кілька екземплярів сервісів спільно використовують сервер і його операційну систему.

Інша перевага цього шаблону полягає в тому, що розгортання екземпляру сервісу відбувається відносно швидко. Потрібно просто скопіювати вихідний код або скомпільований файл сервісу на сервер і запустити його.

Один з основних недоліків полягає в тому, що екземпляри сервісів практично не ізольовані, якщо тільки кожен екземпляр сервісу не є окремим процесом. Хоча ви можете точно здійснювати контроль за використанням ресурсів кожного примірника сервісу, ви не можете обмежувати ресурси, використовувані кожним екземпляром. Неправильно працюючий екземпляр сервісу може використовувати всю пам'ять або ЦП сервера.

Немає ніякої ізоляції сервісів, якщо кілька екземплярів виконуються в одному і тому ж процесі. Всі екземпляри можуть, наприклад, спільно використовувати одну і ту ж купу JVM (для Java додатків). Неправильно працюючий екземпляр сервісу може легко зламати інші сервіси, що працюють в тому ж процесі.

2.3.3.2. Віртуалізація на виділеному сервері

При використанні даного шаблону кожен екземпляр сервісу запускається ізольовано на своєму власному сервісі.

Потрібно упакувати кожен сервіс як образ віртуальної машини (VM), наприклад, такий як Amazon EC2 AMI. Кожен екземпляр сервісу – це віртуальна машина (наприклад, екземпляр EC2), яка запускається з використанням даного образу віртуальної машини.

На рис. 2.11 зображену схема розгортання сервісів на окремих серверах (*хостах*) із використанням VM.

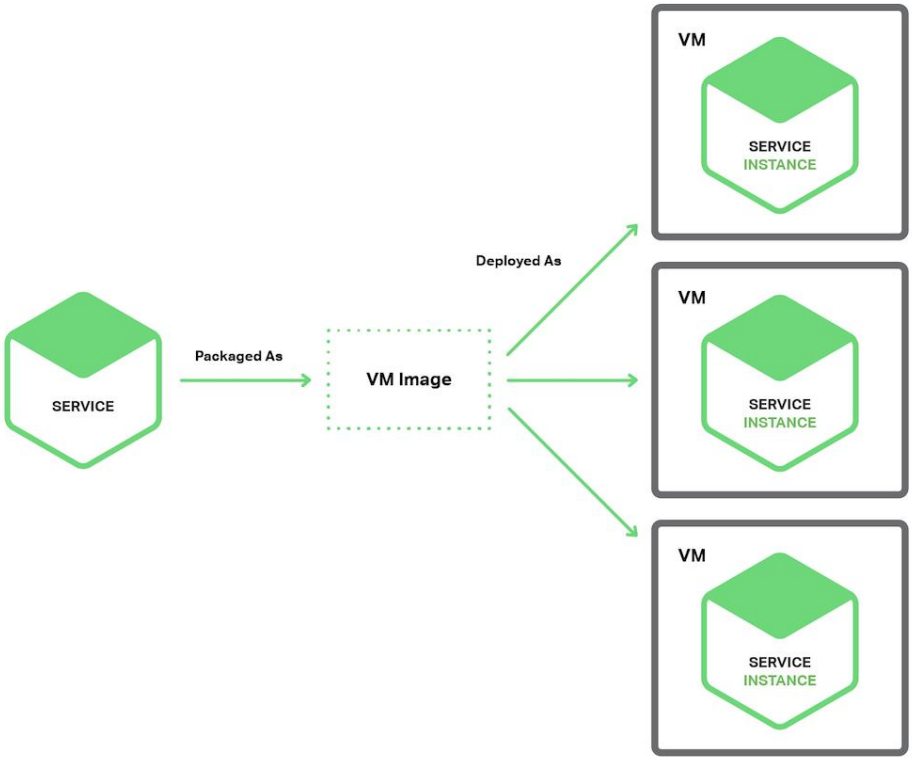


Рис. 2.11. Схема розгортання сервісів із використанням VM

Основною перевагою віртуальних машин є те, що кожен екземпляр сервісу працює в повній ізоляції. Він має фіксований обсяг ресурсів процесора і пам'яті і не може брати ресурси інших сервісів. Ще одна перевага розгортання мікросервісів як віртуальних машин полягає в тому, що можна використовувати розвинену хмарну інфраструктуру. Хмари, такі як AWS,

Azure, Google cloud, надають корисні функції, такі як балансування навантаження і автоматичне масштабування за потребою.

Недоліком є менш ефективне використання ресурсів. Кожен екземпляр сервісу має накладні витрати на віртуальну машину, включаючи операційну систему. Крім того, в типовому загальнодоступному IaaS (*Infrastructure as a Service, Інфраструктура як послуга*) віртуальні машини мають фіксовані розміри, і можливо, що віртуальна машина буде використовуватися недостатньо.

Іншим недоліком даного підходу є те, що розгортання нової версії сервісу зазвичай відбувається повільно. Образи віртуальних машин зазвичай створюються повільно через їх великі розміри. Крім того, для запуску операційної системи зазвичай потрібен деякий час.

2.3.3.3. Контейнеризація на виділеному сервері

При використанні даного підходу, кожен екземпляр сервісу запускається в своєму власному контейнері. Контейнери – це механізм віртуалізації на рівні операційної системи. Контейнер складається з одного або декількох процесів, що виконуються у власному обмеженому середовищі. З точки зору процесів, вони мають свій власний простір імен портів і кореневу файлову систему.

Існує можливість обмеження пам'яті контейнера і ресурсів процесора. Деякі реалізації контейнерів також мають обмеження швидкості I/O. Приклади контейнерних технологій включають Docker і Solaris Zones.

Щоб використовувати дану стратегію, потрібно упакувати сервіс як image (*образ*) контейнера. Образ контейнера – це образ файлової системи, що складається з додатків і бібліотек, необхідних для запуску сервісу.

Можна використовувати менеджер кластерів, такий як Kubernetes або Marathon, для управління контейнерами. Менеджер кластера розглядає хости як пул (групу) ресурсів. Менеджер вирішує, де розмістити кожний контейнер,

ґрунтуючись на ресурсах, необхідних для контейнера, і ресурсах, доступних на кожному хості.

Переваги контейнерів аналогічні перевагам віртуальних машин. Вони ізолюють сервісні екземпляри один від друга. Можна легко відстежувати ресурси, що споживаються кожним контейнером. Також, як і віртуальні машини, контейнери інкапсулюють технологію, використовувану для реалізації ваших послуг. API керування контейнерами також служить API для керування вашими сервісами.

Однак, на відміну від віртуальних машин, контейнери – це «легка» технологія. Контейнерні образи зазвичай дуже швидко створюються. Контейнери також запускаються дуже швидко, так як немає довгого механізму завантаження ОС. Коли контейнер запускається – запускається сервіс.

Хоча контейнерна інфраструктура стрімко розвивається, вона не настільки розвинена, як інфраструктура для віртуальних машин. Крім того, контейнери не так безпечні, як віртуальні машини, оскільки контейнери спільно використовують ядро операційної системи хоста.

2.3.3.4. Безсерверне розгортання

Serverless – безсерверна архітектура додатків. Основу архітектури складають мікросервіси, або функції (lambda), що виконують певне завдання і запускаються на контейнерах. Тобто кінцевому користувачеві дано тільки інтерфейс завантаження коду функції (сервісу) і можливість підключення до функції джерел подій (events).

Переваги даної архітектури:

- відсутність апаратної частини - серверів;
- відсутність прямого контакту і адміністрування серверної частини;
- практично необмежене горизонтальне масштабування;
- оплата тільки за використаний час CPU.

Недоліки:

- відсутність чіткого контролю контейнера – невідомо, де і як запускаються контейнери, хто має доступ;
- відсутність «цілісності» програми: кожна функція – це незалежний об'єкт, що може привести до поганої структурованості.
- «холодний старт» контейнера – перший запуск контейнера з лямбда функцією може становити 2-3 секунди, що не завжди добре сприймається користувачами.

2.4. Масштабування сервісів

Вертикальне масштабування – збільшення потужності кожного компонента системи з метою підвищення загальної продуктивності. Масштабованість в даному контексті означає можливість замінювати апаратне забезпечення в існуючій обчислювальній системі на більш потужні і швидкі компоненти в міру зростання вимог і розвитку технологій. Це найпростіший спосіб масштабування, так як не вимагає ніяких змін в прикладних програмах, що працюють на таких системах.

Горизонтальне масштабування дозволяє запускати декілька екземплярів програми, а не тієї її частини яка вимагають більшого ресурсу, використовуючи балансер навантаження. Якщо є N копій програми, то кожна копія отримує 1 / N усього навантаження. Це простий і часто використовуваний підхід масштабування додатків. Одним з недоліків такого підходу полягає в тому, що, оскільки кожна копія потенційно отримує доступ до всіх даних, щоб бути ефективним, кеш-службі потрібно більше пам'яті. Ще одна проблема цього підходу полягає в тому, що він не враховує розвиток і збільшення складності додатку.

Розглянемо більш детально горизонтальне масштабування, так як дана стратегія є основою стабільності, швидкодії сучасних високонавантажених інформаційних систем. Горизонтальне масштабування побудовано на принципах кластеризації на оркестрації.

2.4.1. Кластеризація

Кластер серверів (*Server Cluster*) – це певна кількість серверів, об'єднаних в групу, які утворюють єдиний ресурс. Дане рішення дозволяє істотно збільшити надійність і продуктивність системи.

Згруповані в локальну мережу декілька комп'ютерів можна назвати апаратним кластером, проте, суть даного об'єднання – підвищення стабільності і працездатності системи за рахунок єдиного програмного забезпечення під управлінням менеджер-модуля (*Cluster Manager*).

На рис. 2.12 зображена схема із трьох серверів і балансера навантаження. Запити від користувачів надходять до LB, який рівномірно їх розподіляє по наявним серверам.

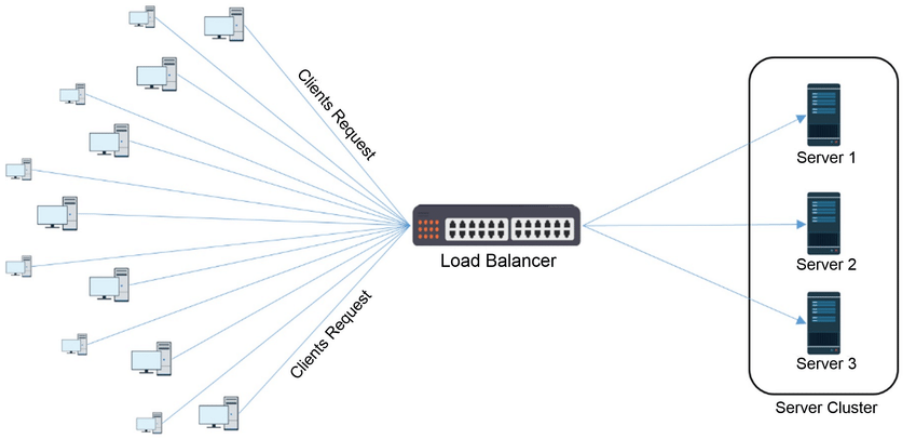


Рис. 2.12. Схема кластеризації серверів

Основні можливості і переваги:

- керування будь-якою кількістю апаратних засобів за допомогою одного програмного модуля;
- можливість додавати і вдосконалювати програмні і апаратні ресурси, без зупинки системи і масштабних архітектурних змін;
- безперебійна робота системи, при виході з ладу одного або декількох серверів;
- ефективний розподіл клієнтських запитів по серверам.

2.4.2. Оркестрація

Оркестрація – це координація взаємодії декількох контейнерів. Оркестрації дозволяє створювати інформаційні системи з безлічі контейнерів, кожен з яких відповідає тільки за одну певну задачу, а спілкування здійснюється через мережеві порти і спільні каталоги. При необхідності кожен такий контейнер можна замінити іншим, що дозволяє, наприклад, швидко перейти на іншу версію бази даних при необхідності.

Існують різні платформи для оркестрації контейнерів, які дозволяють реалізувати зручні та ефективні способи розгортання контейнерних систем, побудови єдиної централізованої служби для застосування політик управління. Найбільш відомі такі системи: Kubernetes, Docker Swarm і Apache Mesos.

Оркестровка – це єдиний централізований виконуваний бізнес-процес (*Orchestrator*), який координує взаємодію між різними службами (рис. 2.13). Orchestrator відповідає за виклик і об'єднання сервісів. Взаємодія між усіма службами описуються в одній кінцевій точці – у композитному сервісі (*composite service).*

Оркестровка включає в себе управління взаємодії і координації між окремими сервісами. Тобто, можна зробити висновок, що оркестрація використовує централізований підхід до правління сервісами.

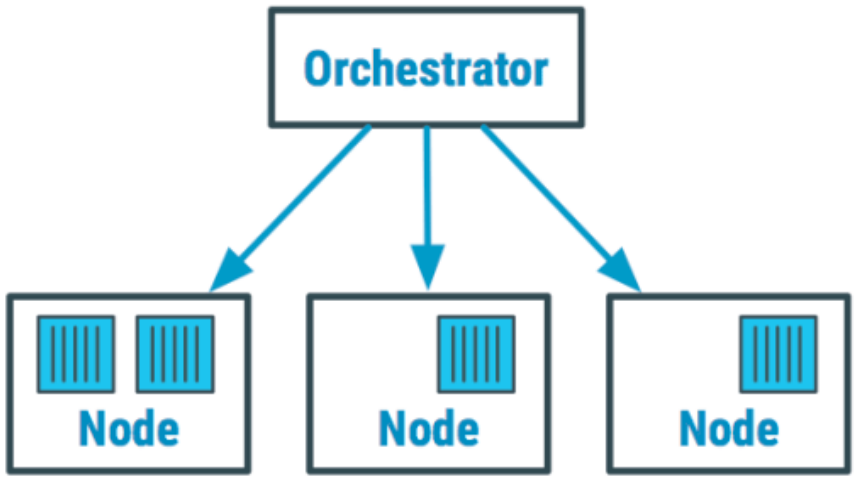


Рис. 2.13. Приклад оркестрації сервісів

2.5. Моніторинг серверів

Розуміння стану інфраструктури та систем важливо для стабільної роботи сервісів. Інформація про працездатність і продуктивності розгортання не тільки допомагає команді вчасно реагувати на проблеми, але і дає їм можливість впевнено вносити всі необхідні зміни. Один з кращих способів отримати цю інформацію – це надійна система моніторингу, яка збирає метрики, візуалізує дані і попереджає операторів, коли щось не працює належним чином.

Метрики – це неформатовані дані про використання ресурсів або поведінку, які можна відстежувати і збирати в системах. Це можуть бути звіти про використання, що надаються операційною системою, або дані більш високого рівня, прив'язані до конкретних функцій або компонентів (наприклад, кількість запитів в секунду, навантаження на CPU, оперативну пам’ять і тд).

Метрики представляють дані про вашу систему, а моніторинг – це процес збору, агрегування та аналізу цих даних для поліпшення розуміння характеристик і поведінки компонентів системи. Дані з різних точок середовища збираються системою моніторингу, яка відповідає за зберігання, агрегацію, візуалізацію даних і автоматичні реагує на зміни, коли значення відповідає заданим умовам.

Метрики корисні, тому що вони дають уявлення про поведінку і працездатність систем, особливо при загальному аналізі. Метрики – це основні значення, що використовуються для розуміння тенденцій, кореляції різних факторів і відстеження змін в продуктивності, споживанні ресурсів або частоті збоїв.

Система сповіщення є компонентом системи моніторингу, яка виконує дії на основі останніх змін метричних значень. Визначення системи оповіщення складаються з двох компонентів: умови (або поріг), заснованого на метриках, і дії, яку потрібно виконати, коли значення виходять за межі прийнятних умов.

Отже, усі метрики можна розділити за категоріями:

- метрики сервера – RAM, CPU, диск і тд;
- метрики додатку – кількість помилок, запитів, швидкодія;
- метрики мережі – затримки мережі, кількість втрачених пакетів;
- метрики кластеру – наявність непрацюючих серверів, навантаження.

Висновок

Мікросервісний підхід, дозволяє легко розробляти, за рахунок невеликих команд розробників для кожного сервісу та масштабувати додатки у залежності від виявленого навантаження та бізнес-вимог, що робить даний шаблон одним із найкращих рішень для бізнесу будь-яких масштабів, на меті у якого стрімкий ріст на розвиток.

Отже, проаналізувавши архітектуру мікросервісів, стратегії побудови та розгортання, можна привести наступні основі переваги і недоліки даного архітектурного підходу:

Переваги:

- ізоляція відмов: великі програми можуть продовжити ефективно працювати, навіть при несправності якогось окремого модуля;
- можливість застосування різного стеку технологій для кожного сервісу;
- модульність бізнес-логіки.

Недоліки:

- мережеві затримки при взаємодії декількох сервісів;
- формати повідомлень – відсутність стандартизації та необхідність узгодження форматів обміну фактично для кожної пари взаємодіючих сервісів призводить до можливих помилок і складнощів налагодження;
- баланс навантаження і відмовостійкості.

РОЗДІЛ 3
DOCKER ЯК ЗАСІБ КОНТЕЙНЕРИЗАЦІЇ МІКРОСЕРВІСІВ

3.1. Концепція контейнеризації

3.1.1. Технологія контейнеризації

Оригінальна технологія контейнерів Linux називається Linux Containers, або LXC. LXC – це метод віртуалізації на рівні ОС призначений для того, щоб запускати безліч ізольованих систем Linux на одному хості.

Контейнери відокремлюють додатки від операційних систем. Це означає, що у користувачів є чиста мінімальна Linux ОС, (або будь-яка інша ОС), і можливість запускати всі процеси в одному або декількох ізольованих контейнерах. Так як операційна система відокремлена від контейнерів, можна переміщати контейнер на будь-який сервер, який підтримує операційну систему контейнера.

Контейнеризація – це «легка» віртуалізація і ізоляція ресурсів, яка дозволяє запускати додаток і необхідний йому мінімум системних бібліотек в повністю стандартизованому контейнері, який взаємодіє з хостом за допомогою певних інтерфейсів, отримуючи доступ до апаратних ресурсів.

Контейнер не залежить від ресурсів або архітектури хоста, на якому він працює.

Всі компоненти, необхідні для запуску програми, упаковуються як один образ і можуть бути використані повторно. Додаток в контейнері працює в ізольованому середовищі і не використовує пам'ять, процесор або диск хостової операційної системи. Це гарантує ізольованість процесів всередині контейнера.

Ще однією перевагою контейнеризації є масштабованість. Можна швидко здійснювати горизонтальне масштабування, створюючи контейнери для короткострокових завдань. З точки зору програми, створення екземпляра образу (контейнера) аналогічно створенню екземпляра процесу, наприклад для служби або веб-додатку. Але для забезпечення надійності при запуску декількох екземплярів одного образу на декількох серверах зазвичай бажано, щоб контейнери (екземпляри образу) виконувалися на різних серверах або віртуальних машинах в різних доменах відмови.

Контейнеризація – це віртуалізація на рівні операційної системи (тобто не апаратна), при якій ядро операційної системи підтримує кілька ізольованих екземплярів простору користувача замість одного.

Простір користувача – це адресний простір віртуальної пам'яті операційної системи, що відводиться для програм користувача.

Ядро (kernel) – центральна частина ОС, що забезпечує додаткам координований доступ до ресурсів комп'ютера, таким як процесорний час, пам'ять, зовнішнє апаратне забезпечення, зовнішній пристрій введення і виведення інформації. Також зазвичай ядро надає сервіси файлової системи і мережевих протоколів.

Контейнери з точки зору користувача повністю ідентичні окремому екземпляру операційної системи. Ядро забезпечує повну ізольованість контейнерів, тому програми з різних контейнерів не можуть впливати один на одного.

Однією з характерних рис підходу на основі контейнеризації є використання всіма контейнерами загального ядра, того ж, що і у хостовій операційній системі. Це дозволяє позбутися від накладних витрат на емуляцію віртуального обладнання та запуску повноцінного екземпляра операційної системи.

Існують реалізації, орієнтовані на створення практично повноцінних екземплярів операційних систем (Solaris Containers, Virtuozzo, OpenVZ), так і

варіанти, які фокусуються на ізоляції окремих сервісів з мінімальним операційним оточенням Jjail, Docker).

Кожен контейнер може вміщати цілий веб-додаток або службу, як показано на рис. 2.1. У даному прикладі вузол Docker – це вузол контейнерів, а *App1*, *App2*, *Svc*1 і *Svc*2 – контейнерні додатки або служби.

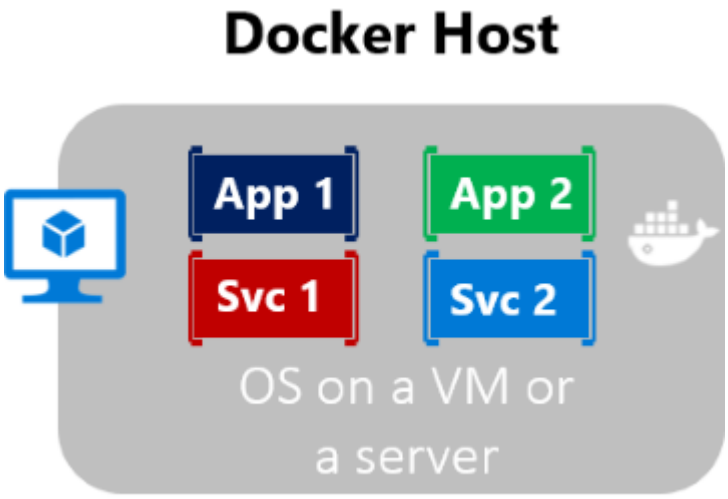


Рис. 3.1. Принцип роботи контейнерів

Гіпервізор (*Hypervisor*) – це монітор віртуальних машин, програма для забезпечення паралельного виконання декількох операційних систем на одному комп'ютері. Гіпервізор забезпечує ізоляцію операційних систем один від одного, розділяє між запущеними ОС ресурси. Одним з таких гіпервізорів є Oracle VirtualBox.

Технологія контейнеризації забезпечує можливість запускати контейнери без використання гіпервізора.

Отже, контейнери надають такі переваги, як ізоляція, переносимість, гнучкість, масштабованість і контроль, протягом усього життєвого циклу програми. Найважливішою перевагою є ізоляція середовища розробки від робочого середовища.

3.1.2. Порівняння технологій віртуалізації і контейнеризації

Існують два основні варіанти віртуалізації, а точніше два підходи до створення незалежних ізольованих обчислювальних просторів на одному фізичному сервері: віртуальні машини, яким потрібен гіпервізор, і віртуальні контейнери. У першому випадку для кожної віртуальної машини використовується власна гостьова ОС, а в другому – для всіх контейнерів застосовується ядро однієї хостової ОС.

Однак, оскільки віртуальні машини включають операційну систему, їх розмір може складати декілька гігабайт. Також недоліком віртуальних машин можна назвати те, що для завантаження ОС і ініціалізації програми, яка в них розміщена, потрібно відносно більше часу.

Контейнери легші і, в основному, їх розмір вимірюється в мегабайтах. Порівнюючи їх продуктивність з віртуальними машинами, контейнери можуть запускатися майже миттєво. При виборі між контейнерами та віртуальними машинами слід враховувати цілі, які потрібно досягти.

Технологія контейнерної віртуалізації

Віртуалізація на рівні ОС дозволяє віртуалізувати фізичні сервери на рівні ядра операційної системи (рис. 3.2).

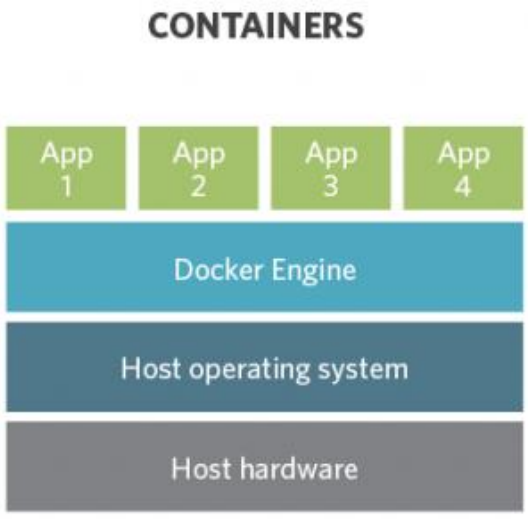


Рис. 3.2. Принцип роботи контейнерної віртуалізації

Шар віртуалізації ОС забезпечує ізоляцію і безпеку ресурсів між різними контейнерами. Шар віртуалізації робить кожен контейнер схожим на фізичний сервер. Кожен контейнер обслуговує тільки додатки в ньому і робоче навантаження.

Основні переваги контейнерної віртуалізації:

- контейнери виконуються на одному рівні з фізичними серверами. Відсутність віртуалізованого обладнання і використання реального обладнання і драйверів дозволяють отримати високу продуктивність;
- кожен контейнер може масштабуватись до ресурсів цілого фізичного сервера;
- технологія віртуалізації на рівні ОС дозволяє домогтися високої щільності: можливе створення і запуск сотень контейнерів на одному фізичному сервері;
- контейнери використовують єдину ОС, що робить їх підтримку і оновлення дуже простим. Додатки можуть бути також розгорнуті в окремому оточенні.

Особливості контейнеризації:

- контейнери виглядають як звичайна Linux-система. Сторонні додатки можуть запускатися в контейнерах без необхідності модифікації;
- користувач може змінювати конфігураційні файли і встановлювати будь-яке додаткове програмне забезпечення в контейнери;
- контейнери повністю ізольовані один від одного (файлова система, процеси, змінні sysctl);
- контейнери використовують динамічні бібліотеки, що значно економить пам'ять;
- контейнери не обмежені одним CPU і можуть використовувати усі CPU хоста.

Технологія апаратної віртуалізації

В апаратній віртуалізації основний компонент – гіпервізор. Дана служба завантажується на сервері і забезпечує взаємодію між апаратним забезпеченням сервера і віртуальними машинами. Щоб надати ресурси віртуальним машинам, забезпечується їх віртуалізація на сервері. Віртуальні машини запускають свою власну копію операційної системи і додатків на віртуалізованому обладнанні (рис.3.3).

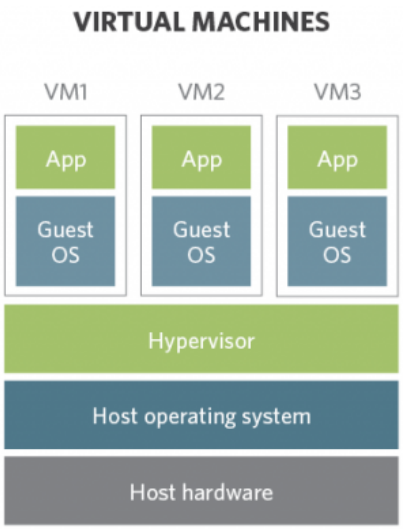


Рис. 3.3. Принцип роботи апаратної віртуалізації

Основні переваги апаратної віртуалізації:

- можливість створення безлічі віртуальних машин з різними операційними системами. Немає залежності від єдиного ядра ОС. Користувач може встановлювати власні патчі на ядро при необхідності отримання розширеної функціональності віртуального сервера;
- віртуальні машини виглядають як звичайний комп'ютер. Вони містять власне віртуальне обладнання та програмне забезпечення, яке може запускатися в віртуальних машинах без необхідності модифікації;
- віртуальні машини повністю ізольовані один від одного і від ОС сервера, де відбувається запуск віртуальних машин (ізоляція на рівні файлової системи, процесів, змінних sysctl);

- завдяки використанню технологій віртуалізації Intel і AMD продуктивність віртуальних серверів дуже висока і наближена до продуктивності реального обладнання.

Кожен віртуальний сервер зберігається як мінімум в двох файлах: файл конфігурації і файл жорсткого диска.

Отже, контейнерна віртуалізація відмінно підходить для більшості завдань на віртуальному Linux-сервері. Буде отримано максимально швидкий час створення сервера і перезавантаження (що дуже важливо при міграції сервера). Також при використанні віртуалізації на рівні ОС є автомасштабування оперативної пам'яті.

Якщо необхідна установка розширень ядра ОС (наприклад для VPN), повний контроль над ядром ОС, можливість використання Docker, ручний апгрейд самої ОС – варто використовувати апаратну віртуалізацію.

3.2. Архітектура Docker

3.2.1. Основні характеристики

Docker – це відкрита платформа для розробки, доставки і експлуатації додатків. Використовуючи контейнери Docker, можна розгортати, копіювати, переносити і робити резервні копії інформації швидше і легше, ніж за допомогою віртуальної машини.

Docker ізолює контейнери, змушуючи їх працювати як єдиний процес. Якщо оточення додатка складається з X одночасних процесів, Docker запустить X контейнерів, кожен зі своїм процесом. На відміну від Docker, LXC контейнери можуть запускати безліч процесів.

Наприклад, для того щоб запустити простий веб-додаток в Docker, знадобиться PHP контейнер (процес php-fpm), Nginx контейнер (процес

веб-серверу), MySQL контейнер (процес бази даних) і декілька контейнерів даних для того, щоб зберігати таблиці БД і іншу інформацію додатків.

У однопроцесорних контейнерів є переваги, наприклад, прості оновлення: не потрібно вбивати процес БД, коли потрібно оновити тільки веб-сервер. Також у однопроцесорних контейнерів ефективна архітектура для того, щоб будувати додатки, на основі мікросервісів.

У своєму ядрі Docker дозволяє запускати практично будь-який додаток, який безпечно ізольований в контейнері. Безпечна ізоляція дозволяє запускати на одному хості декілька контейнерів одночасно. Легка природа контейнера, який запускається без додаткового навантаження гіпервізора, дозволяє оптимальніше використовувати апаратне забезпечення.

Платформа і засоби контейнерної віртуалізації використовуються у таких випадках:

- пакування додатка (і усі залежності) в Docker контейнери;
- доставка упакованих контейнерів командам для розробки і тестування;
- розгортання упакованих контейнерів на сервери: дата-центри, хмари і тд.

3.2.2. Рушій Docker

Docker використовує архітектуру клієнт-сервер. Клієнт Docker взаємодіє з демоном Docker, який береться за себе створення, запуск, масштабування контейнерів, організацію томів, підтримку мережі.

Клієнт і сервер спілкуються через сокет або через RESTful API.

У Docker Engine є три компоненти (рис. 3.4):

- сервер – демон докера під назвою dockerd, який може створювати та керувати образами, контейнерами, мережею тощо;
- REST API – використовується для взаємодії із демоном докера;
- інтерфейс командного рядка (CLI) – клієнт, який використовується для введення Docker команд.

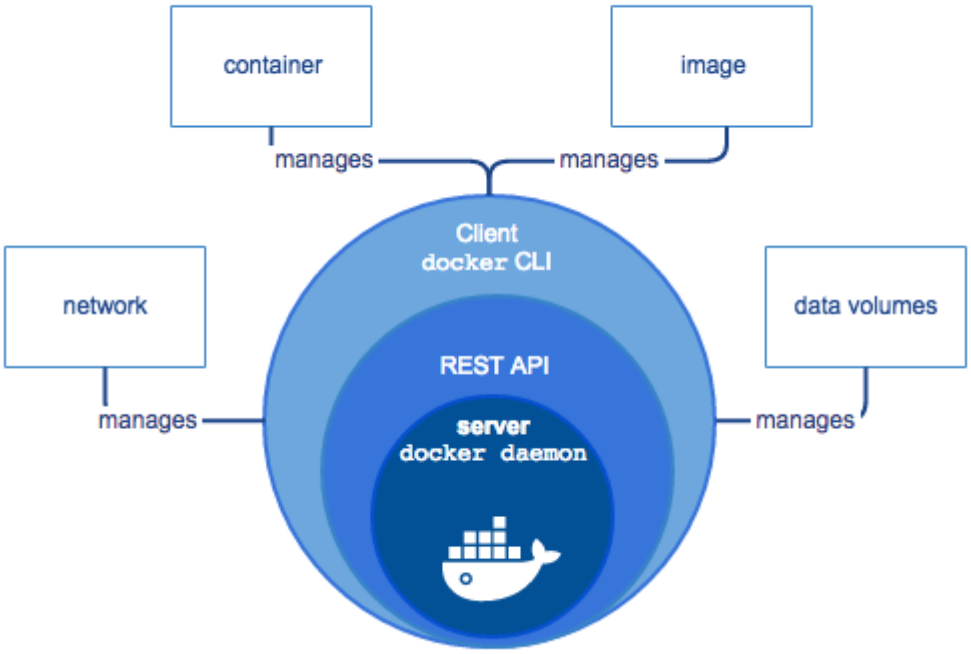


Рис. 3.4. Принцип роботи Docker

3.2.2.1. Docker-демон

Docker-демон відповідає за створення, роботу та моніторинг контейнерів, а також для побудови та зберігання образів.

Демон Докер запускається командою **`docker daemon`** за що зазвичай відповідає ОС хоста. Користувач не взаємодіє з сервером на пряму, а використовує для цього клієнт.

Служба (демон) Docker використовує UNIX-сокет **`/var/run/docker.sock`** для вхідних з'єднань API. Власником даного ресурсу повинен бути тільки користувач root. Зміна прав доступу до даного ресурсу може привезти до небажаного доступу на хостову систему.

Якщо контейнери намагаються використовувати більше пам'яті, ніж доступно системі, може статись Out Of Memory Exception (OOME) і контейнер, або процес docker-демона, може бути знищений OOM ядром системи, що може привести до падіння додатку та втрати важливих даних.

Щоб цього не сталося, потрібно переконатись, що програма працює на хостах із достатньою оперативною пам'яттю.

За допомогою команди dockerd можна запустити сервер Docker. Після введення команди, сервер недійшле інформаційні повідомлення про успішний запуск сервера:

```
$ dockerd
INFO[0000] +job init_networkdriver()
INFO[0000] +job serveapi(unix:///var/run/docker.sock)
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
```

Демон Docker за замовчуванням прослуховує Unix-сокет. Для прослуховування іншого порту, наприклад 2375, можна створити конфігураційний файл `/etc/systemd/system/docker.socket.d/socket.conf` із наступним вмістом:

```
[Socket]
ListenStream=0.0.0.0:2375
```

Демон Docker використовує «драйвер виконання» для створення контейнера. За замовчуванням це власний драйвер Docker runc, однак є підтримка LXC.

Runc відповідає за функціональність:

- *cgroups*, які відповідають за управління використовуваними ресурсами контейнерів (тобто використання CPU та RAM).
- *namespaces*, які відповідають за ізоляцію контейнерів та гарантують, що файлова система, ім'я хосту, мережеве оточення, список процесів та контейнери є відокремленими від інших частин вашої основної операційної системи.

3.2.2.2. Docker-клієнт

Docker-клієнт – головний інтерфейс до Docker. Клієнт отримує команди від користувача і взаємодіє з докер-демоном.

Клієнт Docker розміщує використовується для спілкування з демоном Docker по протоколу HTTP. За замовчуванням відбувається через сокет домену Unix, а саме – IPC (*Inter-Process Communication Socket, сокет міжпроцесорної взаємодії*), але також можна використовувати сокет TCP, що дає можливість створювати віддалених клієнтів.

Так, при роботі з інтерфейсом командного рядка Docker (інтерфейс командного рядка Docker, CLI), в терміналі вводять команди, починаючи з ключів слова docker, що означає звернення до Docker-клієнта. Після чого Docker-клієнт використовує API Docker для відправки команд демону Docker.

3.3. Робота з образами (images)

3.3.1. Створення образів

Образ – це пакет з усіма залежностями і інформацією, яка необхідна для створення контейнера. Образ включає в себе всі залежності (наприклад, платформа, бібліотеки), а також конфігурацію розгортання і виконання для середовища виконання контейнера.

Як правило, образ створюється на основі декількох базових образів, нашарованих один на одного в файлової системі контейнера. Після створення образ залишається незмінним.

Docker образ – це файл, що складається з декількох шарів, у якому зазвичай запаковано код додатку. Усі інструкції створення образу знаходяться у Dockerfile.

Dockerfile – це текстовий файл, що містить інструкції по збірці образу Docker. Це схоже на пакетний сценарій, де перший рядок вказує базовий образ, з якого починається робота, а наступні інструкції встановлюють

необхідні програми, копіюють файли і тд., для створення необхідного робочого середовища.

Образ Docker складається з ряду шарів (*layers*). Кожен шар являє собою інструкцію з Dockerfile. Розглянемо наступний Dockerfile:

```
FROM ubuntu:15.04

COPY . /app

RUN make /app

CMD python /app/app.py
```

Даний Dockerfile містить чотири команди, кожна з яких створює шар. Оператор **FROM** починається зі створення шару з базового образу ubuntu: 15.04. Команда **COPY** додає деякі файли з поточного каталогу вашого клієнта Docker у образ в директорію /app. Команда **RUN** запускає програму за допомогою інструкції make. Нарешті, останній шар **CMD** вказує, яку команду запустити, при старті контейнера.

Кожен шар – це лише сукупність відмінностей від шару перед ним. Шари укладаються один на одного.

Клієнт Docker дозволяє користувачеві ініціювати певні команди, які налаштовують образ Docker:

- *history*: `docker history` показує історію змін шарів образу;
- *tag*: `docker tag` створює тег, тобто версію образа (наприклад, v1.2.7), який дозволяє групувати та впорядковувати образи контейнерів;
- *search*: `docker search` дозволяє шукати образи у репозиторії образів;
- *rmi*: : `docker rmi` видаляє один або декілька образів.

Збірка (пакування) образу – дія по створенню образа на основі інформації та контексту, яка знаходиться у Dockerfile файлі. Збірка образа виконується за допомогою команди `docker build`.

Створюючи новий контейнер, додається новий шар для запису над нижчими шарами. Цей шар часто називають контейнерним шаром. Всі зміни, внесені до запущеного контейнера, такі як запис нових файлів, зміна існуючих файлів та видалення файлів, записуються на цей тонкий шар контейнера. На рис.3.5 показаний контейнер на основі образу Ubuntu 15.04.

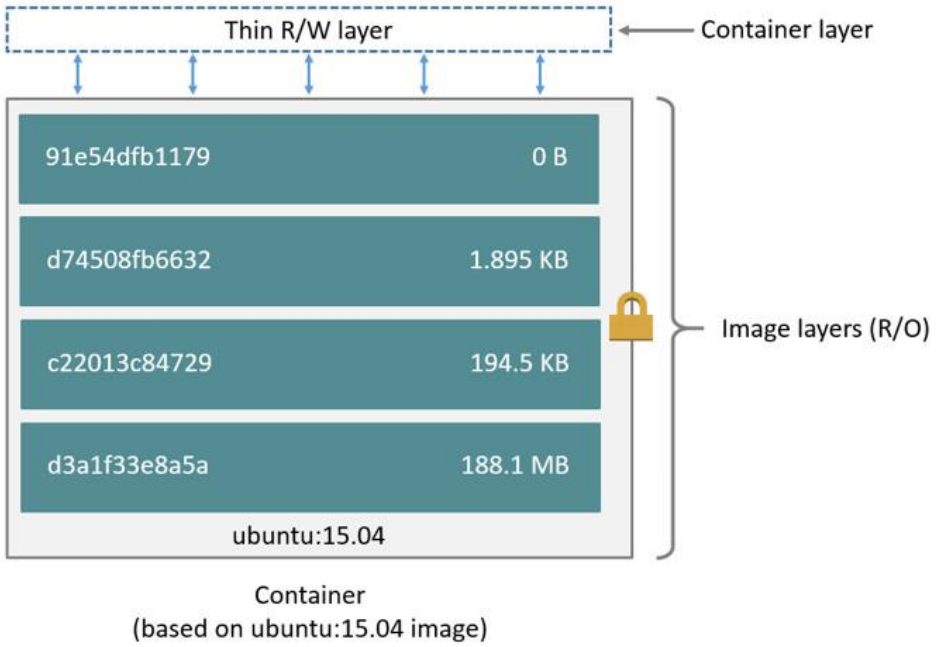


Рис. 3.5. Схема шарів образу

3.3.2. Реєстр образів

Репозиторій – колекція пов'язаних образів Docker, позначена тегом, що вказує на версію образу. Деякі репозиторії містять декілька варіантів одного образу, наприклад образ з пакетом засобів для розробки і тестування (більший обсяг), образ тільки із середовищем виконання (менший обсяг) і тд. Один репозиторій може містити варіанти платформ, наприклад образ Linux і образ Windows.

Реєстр образів – служба, що надає доступ до репозиторіїв. Реєстр за замовчуванням для більшості загальнодоступних образів – Docker Hub. Реєстр зазвичай містить репозиторії декількох команд. Компанії часто

використовують приватні реєстри для зберігання своїх образів і управління ними.

Тобто, Docker-реєстр – це віддалене місце, де містяться всі Docker-образи. Розробники вивантажують туди образи або, навпаки, завантажують їх звідти. Можна використовувати як власний реєстр, так і реєстр будь-якого провайдера, наприклад, AWS або Google Cloud.

Реєстр образів є учасником методології безперервної доставки. Після того як розробник зафіксує свої зміни у системі контролю версій, система CI проведе тести і синтаксичний аналіз коду і, у разі успіху, створить образ і завантажить його у реєстр. Після чого даний образ буде доступним для інших, наприклад, для тестувальників.

3.4. Принцип роботи контейнерів

3.4.1. Запуск контейнерів

Контейнер – це екземпляр образу Docker. Контейнер відповідає за виконання програми, процесу чи служби. Він складається з вмісту образу Docker, середовища виконання та стандартного набору інструкцій. При масштабуванні додатку, можна створити декілька екземплярів контейнера з одного образу.

У контейнерах міститься все, що потрібно для роботи програми. Кожен контейнер створюється з образу. Контейнери можуть бути створені, запущені, зупинені, перенесені або видалені. Кожен контейнер ізольований і є безпечною платформою для додатка.

Контейнер складається з операційної системи, користувацьких файлів і метаданих. Образ Docker включає у себе інформацію як запустити контейнер, тобто, який процес запустити при старті контейнера та інші конфігураційні дані. Docker образ доступний тільки для читання. Коли docker запускає контейнер, він створює рівень для читання / запису зверху образу (використовуючи union file system), в якому може бути запущений додаток.

За допомогою наступної команди можна запустити контейнер:

```
docker run -i -t ubuntu /bin/bash
```

Розглянемо детальніше дану команду. Клієнт запускається за допомогою команди *docker*, з опцією *run*, яка говорить, що буде запущений новий контейнер.

Мінімальна вимога для запуску контейнера є наявність команди яку потрібно виконати коли контейнер буде запущений. У прикладі вище це */bin/bash,* тобто запуск терміналу для роботи у командному рядку всередині контейнера.

Виконання команди *run* включає у себе наступні етапи:

- завантаження образа ubuntu – Docker перевіряє наявність образу ubuntu на локальній машині, і якщо його немає – то завантажує його з Docker Hub. Якщо ж образ є, то використовує його для створення контейнера;
- створення контейнера на базі Docker образу;
- ініціалізація файлової системи і монтування read-write рівня – контейнер щойно створений в файлової системі і read-write шар доданий до образу;
- ініціалізація мережі – створення мережевого інтерфейсу, який дозволяє Docker спілкуватися хост машиною;
- установка IP адреси;
- запуск вказаного процесу – старт програми;
- обробка вихідних даних із додатку – логування стандартного вводу (STDIN), виводу (STDOUT) і потік помилок (STDERR) додатку для того щоб відслідковувати як працює програма.

Тепер запущений робочий контейнер, яким можна управляти для взаємодії із додатком. Всі операції на запис всередині контейнера зберігаються в верхньому шарі (шар контейнеру) і коли контейнер видаляється, верхній

шар, який був доступний для запису, також видаляється, в той час як нижні шари (шари образу) залишаються незмінними.

На рис. 3.6 зображено інформаційний вивід терміну при запуску контейнера. Командою `docker ps` можна перевірити список усіх контейнерів, доступних на хост машині.

```
$ docker run -it ubuntu /bin/bash
root@af588b25a4ad:/#

$ docker ps
CONTAINER ID   IMAGE    COMMAND      CREATED         STATUS         PORTS   NAMES
af588b25a4ad   ubuntu   "/bin/bash"  24 seconds ago  Up 23 seconds          jovial
```

Рис. 3.6. Інформаційний вивід у термінал при створенні контейнера

3.4.2. Кластер контейнерів на базі Docker Swarm

Docker Swarm – це система кластеризації для Docker, яка перетворює набір хостів Docker в один послідовний кластер, так званий Swarm (*рій*).

Кожний хост (*нода*, *node*), в складі такого кластера виступає в якості або керуючої (*manager*) або робочої ноди (*worker*). У кластері повинна бути, як мінімум, одна керуюча нода (*manager*).

Технічно фізичне розташування машин не має значення, однак, бажано мати всі Docker-ноди всередині однієї локальної мережі, в іншому випадку – управління операціями або пошук консенсусу між декількома керуючими нодами може зайняти значну кількість часу.

Основні можливості Docker Swarm:

- Балансування навантаження: Docker Swarm відповідає за балансування навантаження і призначення унікальних DNS-імен, щоб додаток, який розгорнутий в кластері, можна було використовувати так само, якщо б додаток був розгорнутий на одному Docker-хості;

- Динамічне управління ролями: Docker-хости можуть бути додані до Swarm-кластеру без необхідності перезапуску кластера. Більш того, роль вузла (керуючий або робочий) також може динамічно змінюватися;
- Динамічне масштабування сервісів: Кожний сервіс може динамічно масштабуватись як в сторону збільшення екземплярів, так і в бік зменшення. Керуюча нода піклується про додавання або видалення контейнерів на вузлах;
- Контроль відмов: Ноди постійно контролюються керуючої нодою і, якщо будь-яка нода дає збій, то нові завдання запускаються на інших робочих нодах. Docker Swarm також дозволяє створювати декілька керуючих нод для запобігання поломки кластера в разі виходу зі строю єдиною керуючою ноди;
- Rolling-поновлення: оновлення сервісів може застосовуватися поступово;

На рис. 3.7 зображена схема архітектури Docker Swarm.

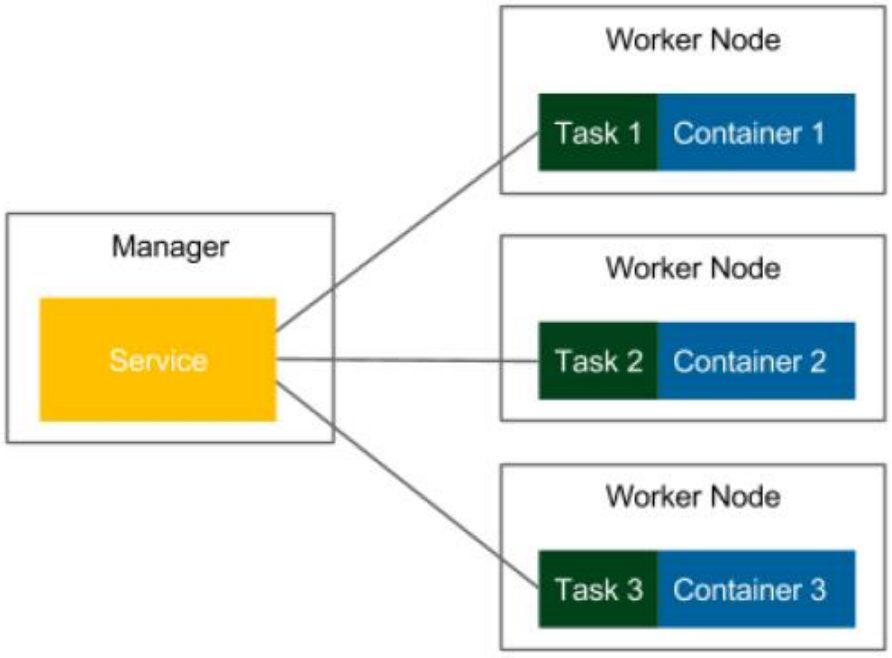


Рис. 3.7. Архітектура Docker Swarm

3.5. Мережа Docker

3.5.1. Взаємодія з іншими контейнерами

При проектуванні розподілених систем працюють з Docker-контейнерами, мережеве взаємодія стає вкрай важливими. Сервіс-орієнтована архітектура, безперечно, спирається на взаємодію між компонентами для коректного функціонування системи в цілому.

При запуску Docker-процесу, створюється новий віртуальний інтерфейс типу «міст» з назвою docker0 в хост-системі. Цей інтерфейс дозволяє Docker створити віртуальну підмережу для використання контейнерами. Міст буде служити основною точкою взаємодії між мережею всередині контейнера і мережею хоста.

Коли Docker запускає контейнер, створюється новий віртуальний інтерфейс і йому призначається адреса в діапазоні підмережі моста. IP-адреса пов'язана із внутрішньою мережею контейнера, надаючи шлях для мережі контейнера до мосту docker0 на системі хоста. Docker автоматично створює правила в iptables для забезпечення переадресації та налаштовує NAT для трафіку з docker0 в зовнішню мережу.

Усі контейнери всередині своєї мережі можуть вільно бачити один одного і спілкуватися між собою, а зовні до них можна достукатися лише прив'язавши контейнерні порти до портів хоста, наприклад, увесь трафік хост-машини з порта 80 можна перенаправляти на будь-який доступний порт контейнера.

Створити мережу для контейнерів можна командною `docker network create mynetwork`. У даному випадку, була створена мережа із назвою `mynetwork`. При створенні контейнерів можна вказувати, що вони повинні бути підключені до даної мережі: `docker run --net=mynetwork --name=web1`. Тепер, контейнери в одній мережі `mynetwork` можуть спілкуватись між собою використовуючи імена контейнерів, а Docker DNS, візьме на себе пошук ip-адреси потрібного контейнера.

3.5.2. Варіанти організації мережевого оточення

Мережа Docker побудована на Container Network Model (CNM), яка дозволяє будь-кому створити свій власний мережевий драйвер. Таким чином, у контейнерів є доступ до різних типів мереж і вони можуть підключатися до декількох мереж одночасно. Крім різних сторонніх мережевих драйверів, у самого Docker є 4 вбудованих:

- Bridge – в даній мережі контейнери запускаються за замовчуванням. Зв'язок встановлюється через bridge-інтерфейс на хості. У контейнерів, які використовують однакову мережу, є своя власна підмережа, і вони можуть передавати дані один одному за замовчуванням;
- Host – даний драйвер дає контейнеру доступ до власного простору хоста (контейнер буде бачити і використовувати той же інтерфейс, що і хост);
- Macvlan – драйвер дає контейнерам прямий доступ до інтерфейсу і суб-інтерфейсу (vlan) хоста.
- Overlay – даний драйвер дозволяє будувати мережі на декількох хостах з Docker (зазвичай на Docker Swarm кластері). У контейнерів також є свої адреси мережі і підмережі, і вони можуть безпосередньо обмінюватися даними, навіть якщо вони розташовуються фізично на різних хостах.

Найчастіше використовуються bridge драйвер. Можна створити свої власні bridge-мережі за допомогою команди `docker network create`, вказавши опцію `--driver bridge`.

Наприклад, команда `docker network create --driver bridge --subnet 192.168.100.0/24 --ip-range 192.168.100.0/24 my-bridge-network` створює bridge-мережу з ім'ям `my-bridge-network` і підмережею `192.168.100.0/24`.

Контейнери на одному хості здатні здійснювати доступ до сервісів, хост-система просто буде направляти запити до інтерфейсу docker0 в відповідне місце.

Контейнери можуть відкривати свої порти для хоста, на які вони можуть приймати трафік, що приходить із зовнішнього світу. Відкриті порти можуть бути відображені (mapped) на хост-систему або шляхом вибору конкретного порту, або дозволом Docker вибрати випадковий невикористовуваний порт. У цих випадках Docker подбає про всіх правилах переадресації та конфігурації iptables для коректної маршрутизації пакетів.

На рис.3.8 зображена схема мережевої архітектури Docker із двома ізольованими мережами контейнерів.

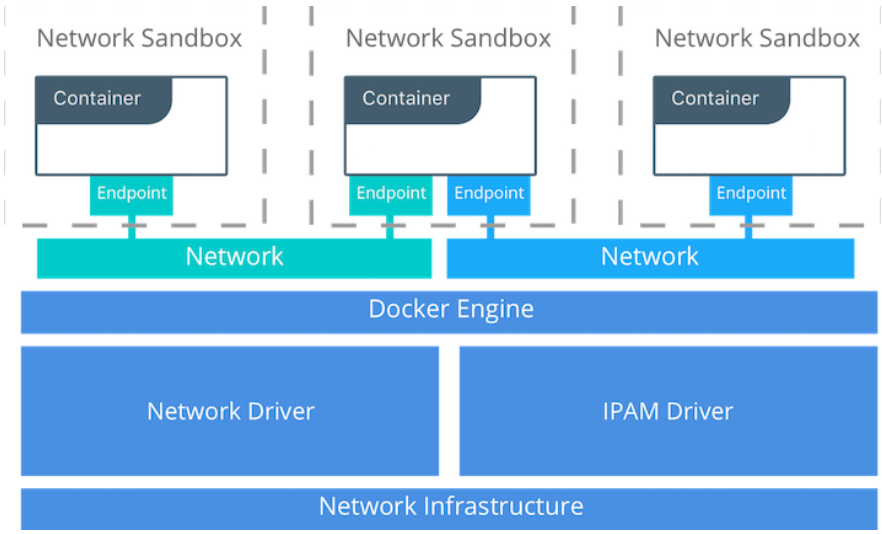


Рис. 3.8. Архітектура мережі Docker

3.6. Збереження персистентних даних за допомогою томів

Docker томи (*volumes*) – файлова система з можливістю запису, яку може використовувати контейнер. Оскільки образи доступні тільки для читання, а більшості програм потрібно можливість запису в файлову систему, тому додають шар з підтримкою запису поверх образу контейнера, який програми зможуть використовувати як файлову систему з можливістю запису.

Том – це файлова система, яка розташована на хост-машині за межами контейнерів. Створенням і управлінням томами займається Docker рушій.

Основні властивості томів Docker:

- являють собою засоби для постійного зберігання інформації;
- самостійні і відокремлені від контейнерів;
- можуть бути спільно використовувані різним контейнерами;
- дозволяють організувати ефективне читання і запис даних;
- томи можна розміщувати на ресурсах віддаленого хмарного провайдера;
- можливість шифрування.

Створити том можна наступною командою: `docker volume create --name my_volume`.

Декілька контейнерів можуть спільно використовувати один або кілька томів даних. Тим не менш, якщо декілька контейнерів пишуть в один і той же загальний том, це може привести до пошкодження даних. Перевага використання спільних томів полягає в тому що вони не залежать від хоста. Це означає що том може бути доступний на будь-якому хості. Томи даних безпосередньо доступні з Docker хоста, тобто можна зчитувати і записувати дані за допомогою стандартних інструментів Linux.

Висновок

Docker розроблений для більш швидкого розгортання додатків. За допомогою Docker можна відокремити додаток від інфраструктури і розгортати додаток на будь-якому хості, який підтримує Docker.

Docker допомагає розгортати, швидше тестувати, зменшити час між написанням коду і запуску коду. Docker робить це за допомогою «легкої» платформи контейнерної віртуалізації.

За допомогою Docker можна запускати будь-який додаток, який безпечно ізольований у контейнері.

РОЗДІЛ 4
РОЗРОБКА СИСТЕМИ КОНТЕЙНЕРИЗАЦІЇ МІКРОСЕРВІСІВ

4.1. Аналіз вимог

Архітектура на базі мікросервісів будується на основі атомарних, модульних сервісів, які виконують обмежену кількість задач.

Отже, базовою вимогою є незалежність сервісів один від одного.

Задача стандартизації взаємодії мікросервісів один з одним буде вирішена за допомогою специфікації JSON API.

JSON API – це специфікація інтерфейсу взаємодії мій клієнтом і сервером. JSON API призначений для мінімізації як кількості запитів, так і кількості переданих даних між клієнтами та серверами. Дана ефективність досягається без шкоди для читабельності, гнучкості чи відкритості запитів та відповідей.

За допомогою REST підходу буде вирішено завдання стандартизації інтерфейсу керування ресурсами.

REST – це стиль архітектури програмного забезпечення для розподілених систем, який, як правило, використовується для побудови веб-служб. Системи, що підтримують REST, називаються RESTful-системами.

У загальному випадку REST є дуже простим інтерфейсом управління інформацією без використання додаткових внутрішніх прошарків. Кожна одиниця інформації однозначно визначається глобальним ідентифікатором, таким як URL. Кожна URL в свою чергу має строго заданий формат.

Проблему часових затрат на розгортання та збірку сервісів буде вирішено за допомогою концепції **безперервної доставки**. Підхід безперервної доставки допомагає швидше та зручніше збирати додатки, а саме образи, застосовуючи при цьому допоміжні засоби , такі як, автоматизовані тести, перевірка синтаксису коду і тд.

Проблему відмовостійкості та масштабованості буде вирішено за допомогою використання концепції **кластеризації**, що дозволить звести до мінімуму час простою додатку із-за деяких нештатних ситуацій та оптимально розподіляти навантаження між серверами.

Для контролю над станом серверів необхідно реалізувати **моніторинг** системи, а саме контроль над RAM, CPU, об’єм вільного місця на диску, виявлення помилок.

Отже, можна виділити такі вимоги:

- забезпечення атомарності сервісу;
- стандартизація формату обміну повідомленнями між мікросервісами;
- стандартизація інтерфейсу управління інформацією;
- оптимізація часу збірки та розгортання;
- забезпечення відмовостійкості та масштабованості;
- організація моніторингу системи.

4.2. Використані технічні і програмні засоби

Docker

Docker – це програмна платформа для швидкої розробки, тестування і розгортання додатків. Docker упаковує ПО в стандартизовані блоки, які називаються контейнерами.

Використана версія Docker – 19.03.5.

PHP

PHP – скриптова мова загального призначення, інтенсивно застосовується для розробки веб-додатків.

Використана версія PHP – 7.2.26.

Laravel

Laravel – безкоштовний веб-фреймворк з відкритим кодом, призначений для розробки веб-додатків з використанням архітектурної моделі MVC (*Model View Controller, модель-уявлення-контролер*). Laravel має вбудовані модулі для роботи із БД, HTTP запитами, сесіями.

Використана версія Laravel – 6.10.1.

PHPSTORM

PhpStorm – комерційне крос-платформне інтегроване середовище розробки для PHP.

PhpStorm включає у себе інтелектуальний редактор для PHP, HTML і JavaScript з можливостями аналізу коду на льоту, запобігання помилок в коді і автоматизованими засобами рефакторингу для PHP і JavaScript. Є повноцінний SQL-редактор з можливістю редагування отриманих результатів запитів.

Використана версія PHPSTORM – 2019.3.

MySQL

MySQL – це система керування базами даних, а саме реляційними БД. У реляційній базі даних дані зберігаються в окремих таблицях, завдяки чому досягається виграш в швидкості і гнучкості.

SQL як частина системи MySQL можна охарактеризувати як мову структурованих запитів плюс найбільш поширений стандартний мова, яка використовується для доступу до баз даних.

Використана версія MySQL – 5.7.21.

VirtualBox

VirtualBox – це програмний продукт віртуалізації для різних операційних систем. Тобто, це програмне забезпечення, яке імітує справжній комп'ютер, що дає можливість користувачеві встановлювати, запускати і використовувати інші операційні системи, як звичайні додатки. Такий собі комп'ютер в комп'ютері.

Використана версія VirtualBox – 6.1.2.

Prometheus

Програмне забезпечення для збору і аналізу показників стану веб-додатків, а саме RAM, CPU, кількість помилок і інші користувацькі метрики.

Використана версія Prometheus – 2.15.2.

Grafana

Інструмент для візуалізації метрик, що дозволяє будувати гнучкі графіки, діаграми, проводити обчислення над даними метрик.

Використана версія Grafana – 6.5.3.

4.3. Контейнеризація мікросервісів

Docker дозволяє запаковувати сервіси з усіма залежностями в образи, що дає можливість передавати дані образи через мережу і запускати на будь-якому сервері, де встановлене програмне забезпечення Docker.

Для реалізації постановки задачі створимо декілька мікросервісів: *api-gateway, users-api, documents-api, statistics-api.*

API Gateway надає зовнішнім клієнтам загальний інтерфейс взаємодії з системою. Також даний шлюз нерідко виступає в якості єдиної точки входу для зовнішніх запитів. В такому випадку на шлюз може бути покладена відповідальність за забезпечення безпеки транспортного рівня з використанням різних каналів безпеки.

До переваг використання *api-gateway* підходу також відносять: ізоляція клієнтів від структури сервісів, уніфікований API для клієнтів, автоматичне розпізнавання місцезнаходження потрібно сервіса.

Задача мікросервіса *users-api* – зберігання інформації про користувачів.

Задача мікросервіса *documents-api* –інформація про документи.

Задача мікросервіса *statistics-api* – агрегація статистики відвідувань.

Кожен мікросервіс має свою власну БД MySQL.

На рис. 4.1. зображена схема архітектури проектованого додатку.

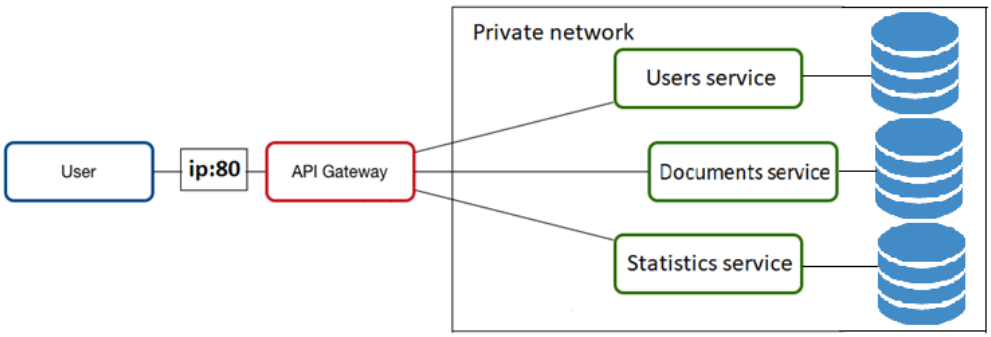


Рис. 4.1. Архітектура додатку

4.3.1. Розробка мікросервісів

4.3.1.1. Мікросервіс api-gateway – єдина точка входу

В основі сервіса *api-gateway* знаходиться веб-сервер *nginx*, який прослуховує по порт 80 (порт HTTP) і приймає запити від користувачів (веб-браузер, мобільний додаток і тд.). На основі URL вирішується, якому мікросервісу перенаправити запит.

Код сервіса розміщений у публічному репозиторії за посиланням: https://github.com/zhenia97/api-gateway.

URL, що починається із `api/v1/users` – перенаправляється до мікросервіса users-api, `api/v1/documents` – перенаправляється до мікросервіса

documents-api, `api/v1/statistics` – перенаправляється до мікросервіса statistics-api.

Конфігурація веб-сервера знаходить у файлі default.conf, який аналізується *nginx* при запуску служби. Інструкції конфігурації веб-сервера приведено у додатку С.

4.3.1.2. Мікросервіс users-api

Мікросервіс *user-api* включає у себе веб-сервер *nginx* і службу *php-fpm* для обробки php-скриптів.

Код сервіса розміщений у публічному репозиторії за посиланням: https://github.com/zhenia97/users-api.

На рис.4.2 зображений REST-сумісний інтерфейс для взаємодії із сервісом. Тобто, GET запит на URL адресу `[app_ip]:80/api/v1/users` дає змогу отримати користувачів додатку. На рис. 4.3 зображено приклад відповіді за даним запитом.

```
+--------+----------+------------------+------+-----------------------------------------+------------+
| Domain | Method   | URI              | Name | Action                                  | Middleware |
+--------+----------+------------------+------+-----------------------------------------+------------+
|        | GET|HEAD | api/v1/users     |      | App\Http\Controllers\UsersController@index   |            |
|        | GET|HEAD | api/v1/users/{id} |      | App\Http\Controllers\UsersController@getById |            |
+--------+----------+------------------+------+-----------------------------------------+------------+
```

Рис. 4.2. Інтерфейс взаємодії із сервісом users-api


```
{
    "data": [
        {
            "id": 1,
            "name": "magnus.bashirian",
            "email": "gislason.loy@gmail.com",
            "created_at": "2020-01-19 13:22:17",
            "updated_at": null
        },
    ],
    "meta": {
        "container_id": "6ae6484f505a",
        "service_name": "users-api"
    }
}
```


Рис. 4.3. Приклад відповіді на API запит

Збірка образу виконується за допомогою Dockerfile, який містить у собі усі інструкції по тому, як збирати кожен шар образу. Кожна інструкція додає до образу новий шар, тобто варто мінімізувати їх кількість, щоб зменшити розмір образу.

У додатку В приведено вміст Dockerfile файлу для сервісів *users-api*, *documents-api*, *statistics-api*.

Інструкція FROM вказує, який базовий образ використовувати, у даному випадку це образ із встановленим php. Нижче даної інструкції йде встановлення усіх необхідних пакетів.

Інструкція CMD вказує на запуск служб додатку: *nginx* та *php-fpm*.

4.3.1.3. Мікросервіс documents-api

Мікросервіс *documents-api* включає у себе веб-сервер *nginx* і службу *php-fpm* для обробки php-скриптів.

Код сервіса розміщений у публічному репозиторії за посиланням: https://github.com/zhenia97/documents-api.

На рисунку 4.4. зображено REST-сумісний інтерфейс взаємодії із сервісом *documents-api*. Тобто, за допомогою визначених GET запитів можна дізнатись інформацію про документи.

```
+--------+----------+--------------------------+------+-----------------------------------------------------+------------+
| Domain | Method   | URI                      | Name | Action                                              | Middleware |
+--------+----------+--------------------------+------+-----------------------------------------------------+------------+
|        | GET|HEAD | api/v1/documents         |      | App\Http\Controllers\DocumentsController@index      |            |
|        | GET|HEAD | api/v1/documents/user/{id} |      | App\Http\Controllers\DocumentsController@getByUserId |            |
|        | GET|HEAD | api/v1/documents/{id}    |      | App\Http\Controllers\DocumentsController@getById    |            |
+--------+----------+--------------------------+------+-----------------------------------------------------+------------+
```

Рис. 4.4. Інтерфейс взаємодії із сервісом documents-api

4.3.1.4. Мікросервіс statistics-api

По аналогії із іншими сервісами, мікросервіс *statistics-api* включає у себе веб-сервер *nginx* і службу *php-fpm* для обробки php-скриптів.

Код сервіса розміщений у публічному репозиторії за посиланням: https://github.com/zhenia97/statistics-api.

На рисунку 4.5. зображено REST-сумісний інтерфейс взаємодії із сервісом *statistics-api*. Тобто, за допомогою GET запитів можна отримати статистику відвідувань по кожному користувачу.

```
+--------+----------+---------------------------+------+-----------------------------------------------------+------------+
| Domain | Method   | URI                       | Name | Action                                              | Middleware |
+--------+----------+---------------------------+------+-----------------------------------------------------+------------+
|        | GET|HEAD | api/v1/statistics         |      | App\Http\Controllers\StatisticsController@index     |            |
|        | GET|HEAD | api/v1/statistics/user/{id} |      | App\Http\Controllers\StatisticsController@getByUserId |            |
+--------+----------+---------------------------+------+-----------------------------------------------------+------------+
```

Рис. 4.5. Інтерфейс взаємодії із сервісом statistics-api

4.3.2. Стандартизація формату повідомлень

За допомогою специфікації JSON API було стандартизовано формат повідомлень для обміну між мікросервісами.

Для усіх API відповідей від сервісів, слідуючи стандарту, було додано заголовок: *Content-Type: application/json; charset=UTF-8.*

Була створена бібліотека «JsonApi» на мові PHP для уніфікації формату повідомлень, що дало змогу спростити підтримку сервісів і покращило модульність коду. Усі повідомлення були стандартизовані у формат виду:

```
[
      'data' => [],
      'meta' => [],
      'errors' => [],
 ]
```

Де, в ключ *data* записується інформація про запитуваний ресурс, *meta* – додаткова, довідкова інформація про ресурс, *errors* – інформація у разі виникнення помилки.

Код бібліотеки був розміщений на публічному репозиторії GitHub за посиланням: https://github.com/zhenia97/jsonApi. Приклад використання бібліотеки приведено нижче.

```
public function index(): JsonResponse
{
    $data = new DataObject(
        AppUserModel::all()->toArray()
    );
    $meta = new MetaObject([
        'container_id' => getenv('HOSTNAME'),
        'service_name' => self::SERVICE_NAME,
    ]);
    $jsonApi = new JsonApi($data, $meta);
    return response()->json(
        $jsonApi->toArray(), 200, [], JSON_PRETTY_PRINT
    );
}
```

4.3.3. Версіонування образів

Для фіксації змін у коді сервісів було застосовано підхід семантичного версіонування. Даний підхід допомагає попереджувати ситуації із можливою несумісністю служб, особливо коли йде мова про десятки взаємопов'язаних сервісів.

Загальною практикою при реалізації версіонування API є включення номера версії в URL-адресу виклику API-методу. Наприклад, `[app_ip]:80/api/v1/users` означає виклик API-методу сервісу *users-api* із версією 1.

Версіонування також було застосовано до Docker образів, наприклад, один із перших образів мікросервісу *users-api* був позначений тегом (версією) *users-api:1.0.0.*

Версія образу із тегом *latest*, наприклад, *users-api:latest*, означає що використовується нестабільна версія, яка ще у процесі розробки.

4.3.4. Аналіз тривалості збірки образів

Було проаналізовано тривалість збірки кожного мікросервісного образу. Із табл. 4.1 видно, що сервіс *api-gateway*, має мінімальне значення, це пояснюється тим, що кодова база даного образу мінімальна.

Отже, можна зробити висновок, що тривалість збірки є прямопропорційною величиною до розміру образу, тому для швидкого розгортання та збірки потрібен мінімальний розмір образу.

Таблиця 4.1

Аналіз тривалості збірки образів

Сервіс	Кількість вимірів	Розмір образу (мегабайт)	Середня тривалість збірки (секунд)
users-api	20	204,7	195,4
documents-api	20	201,4	191,8
statistics-api	20	203,1	194,3
api-gateway	20	48,5	54,2

4.4. Реалізація безперервної доставки

Система реєстру образів Docker Hub може автоматично створювати образ із вихідного коду у зовнішньому сховищі (наприклад, GitHub) та автоматично пересилати упакований образ у репозиторій.

Під час налаштування автоматизованих збірок, потрібно створити список гілок та тегів, які потрібно упакувати. Після того як нова версія коду потрапляє у систему контролю версій, DockerHub автоматично запускає збірку образу на базі вихідного коду. Після того як збірка завершена, образ поміщується у репозиторій із присвоєним тегом.

Також існує можливість встановити значення змінних середовища, які використовуються у процесах збірки, що дає змогу динамічно виконувати ті чи інші інструкції при збірці. У випадку, якщо збірка завершилась із помилкою, існує можливість запустити процес знову.

На рис. 4.6 зображено налаштовану конфігурацію автоматизованих збірок образів.

Завдяки опції кешування (*Build Caching*) досягається зменшення тривалості збірки за рахунок збереження проміжних шарів образу. Кожен шар образу поміщається сховище DockerHub і при наступній збірці, якщо хеш-сума поточного шару не змінилась, використовується шар із сховища, замість того щоб створювати новий.

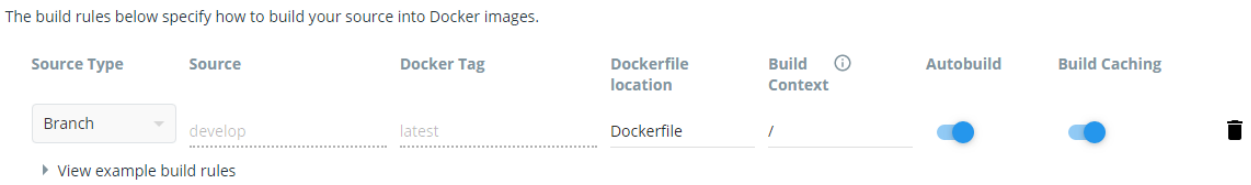


Рис. 4.6. Конфігурація автоматизованих збірок

Підхід на основі безперервної доставки дав змогу автоматизовано створювати версії Docker образів без особистого запуску збірки.

4.5. Забезпечення відмовостійкості додатку

Відмовостійкі кластери широко використовуються для підтримки важливих БД, для зберігання файлів у мережах, бізнес-пропозицій та систем обслуговування клієнтів, таких як веб-сайти електронної комерції.

Наведемо основі поняття, які застосовуються при проектуванні відмовостійкої інфраструктури.

Відмовостійкість (*Fault Tolerance, FT*) – здатність системи до подальшої роботи після виходу з ладу будь-якого її елемента.

Кластер – група серверів (обчислювальних одиниць), об'єднаних каналами зв'язку.

Відмовостійкий кластер (*Fault Tolerant Cluster, FTC*) – кластер, відмова сервера у якому, не приводить до повної непрацездатності всього кластеру. Задачі виведеної з ладу машини розподіляються між однією або декількома іншими нодами в автоматичному режимі.

Неперервна доступність (*Continuous Availability, CA*) – концепція при якій, користувач може в будь-який момент використовувати службу, без переривань, навіть у разі відмови одного або декількох вузлів.

Висока доступність (*High Availability, HA*) – у випадку виходу з ладу служби, система автоматично відновить працездатний стан, час простою мінімізується.

4.5.1. Проектування кластерної інфраструктури

Перевага кластеризації для підвищення доступності стає очевидною у разі збою будь-якого вузла, при чому інший вузол кластера може взяти на себе навантаження несправного вузла, і, таким чином, користувачі не помітять переривання в доступі до сервіса. Кластеризація може бути здійснена на різних рівнях комп'ютерної системи, включаючи апаратне забезпечення, операційні системи, програми-утиліти, системи управління і додатки. Чим більше рівнів системи об'єднані кластерної технологією, тим вище надійність, масштабованість і керованість кластеру.

На рис. 4.7 зображено проектовану кластерну архітектуру. Єдиною точкою входу у додаток є сервіс *api-gateway*: сервер *nginx* із відкритим портом 80. Тобто додаток знаходиться у приватній мережі, недоступній ззовні, крім порту 80 (HTTP). У свою чергу, екземпляри мікросервісів, у кількості трьох на кожен сервіс, теж знаходяться у приватній мережі.

Три балансери навантаження виступають єдиною точкою входу до мікросервісів: запит потрапляє на балансер навантаження, після чого перенаправляється на потрібний, зазвичай менш навантажений у даний момент, екземпляр.

Зв'язок між мікросервісами досягається за рахунок DNS імен, які присвоєні кожному балансеру навантаження, тобто, якщо із сервіса *users-api* потрібно відправити запит до сервіса *documents-api*, потрібно використовувати DNS ім’я балансера навантаження сервіса *documents-api*.

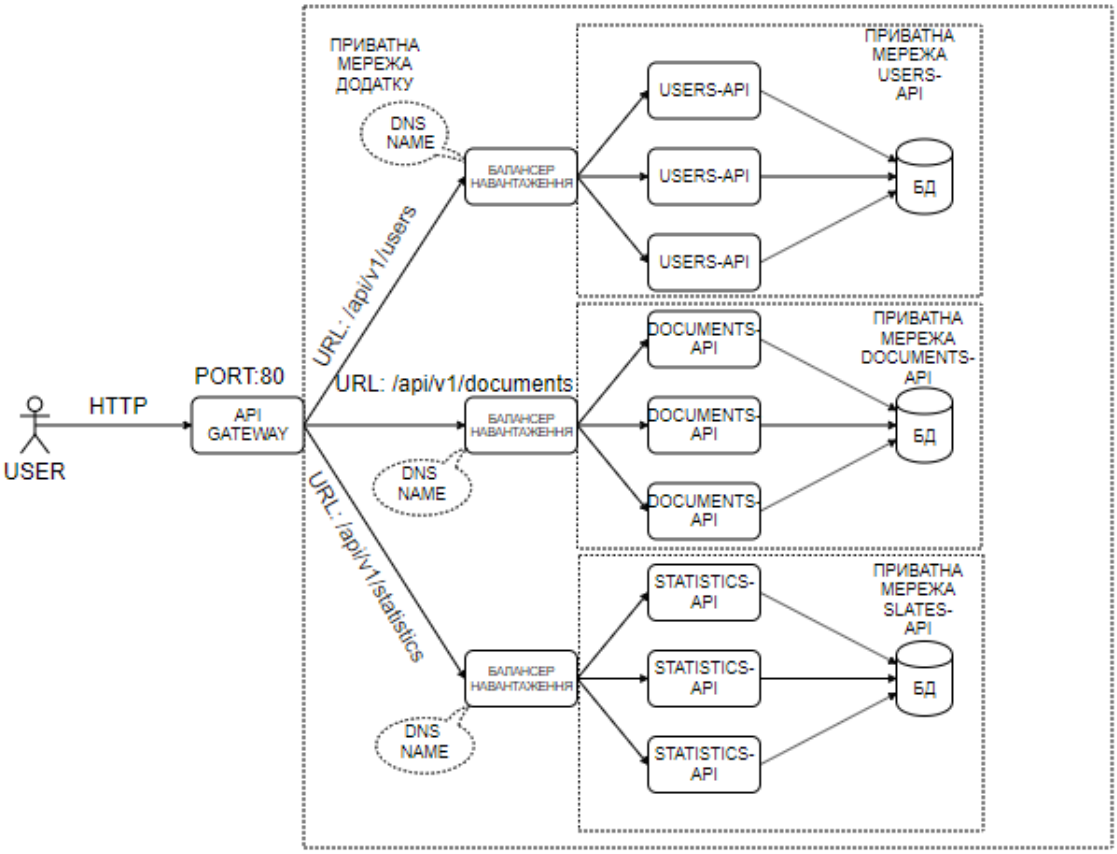


Рис. 4.7. Проектована кластерна інфраструктура

4.5.2. Створення масштабованого кластеру мікросервісів

Використовуючи технологію Docker Swarm, яка вбудована в Docker, можна створити масштабований кластер на базі декількох хостів.

Використовуючи VirtualBox, було створено чотири віртуальні машині (рис. 4.8), які будуть об’єднані в кластер. Машина із назвою *default* буде виступати у якості менеджера, тобто головної ноди, яка буде керувати кластером і збалансовувати навантаження між іншими нодами.

```
C:\Users\Evgeniy>docker-machine.exe ls
NAME             ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER     ERRORS
default          *        virtualbox   Running   tcp://192.168.99.100:2376           v19.03.5
swarm-worker     -        virtualbox   Stopped                                       Unknown
swarm-worker-2   -        virtualbox   Stopped                                       Unknown
swarm-worker-3   -        virtualbox   Stopped                                       Unknown
```

Рис. 4.8. Список віртуальних машин

Запуск кластера відбувається за допомогою команди, де `swarm-docker-compose.yml` – конфігураційний файл кластеру:

```
docker stack deploy --compose-file swarm-docker-compose.yml app
```

Тривалість розгортання кластеру: 7.31 секунди.

У додатку А приведений вміст конфігураційного файлу кластера.

Docker Swarm автоматично займається розподіленням сервісів, і їх екземплярів у вигляді контейнерів, по чотирьом запущеним віртуальним машинам, які є вузлами кластеру.

Із виводу команди (рис. 4.9) видно, що сервіси *users-api, documents-api, statistics-api* бути створені у кількості три контейнери на кожен сервіс. Сервіс api-gateway та БД кожного сервіса були створені одиничними контейнерами.

```
docker@default: $ docker service ls
ID                  NAME                MODE                REPLICAS  IMAGE                                  PORTS
r8mazuiasx87        app_api-gateway     replicated          1/1       zhenia97chap/api-gateway:latest        *:80->80/tcp
qk93dfs6qpps        app_documents-api   replicated          3/3       zhenia97chap/documents-api:latest
ooapjb9mdzbj        app_documents-db    replicated          1/1       mysql:5.7.21
cidkfcovzlnp        app_statistics-api  replicated          3/3       zhenia97chap/statistics-api:latest
y65uludclat8        app_statistics-db   replicated          1/1       mysql:5.7.21
nxrpk81hhzcl        app_users-api       replicated          3/3       zhenia97chap/users-api:latest
biiqm82dej2h        app_users-db        replicated          1/1       mysql:5.7.21
```

Рис. 4.9. Список сервісів кластеру

Масштабування кластеру відбувається за допомогою команди: `docker service scale app_users-api=5`

Тривалість масшабування сервісу: 3.89 секунд.

У даному випадку відбувалось масштабування сервісу *users-api* із трьох контейнерів до п’яти.

Коли необхідно зупинити робочу ноду, наприклад, для обслуговування або просто видалити її з кластера, можна використовувати Swarm draining-нод. Перехід статусу ноди зі стану Active в стан Drain призводить до того, що керуюча нода переміщує всі запущені завдання / контейнери з drain-ноди на інші активні ноди шляхом зупинки таких контейнерів на drain-ноді і запуску їх на active-нодах.

Наприклад, щоб перевести ноду swarm-worker у режим обслуговування, потрібно ввести команду:

```
docker node update --availability drain swarm-worker
```

Статус ноди «керуюча» або «робоча» можна змінювати. Docker Swarm дозволяє використовувати дві і більше керуючих ноди. Для досягнення більшої відмовостійкості в разі виходу зі строю однієї керуючої ноди рекомендується використовувати дві або більше керуючих нод.

Наприклад, наступна команда переодить ноду *swarm-worker* із «робочої» у «керуючу»: `docker node promote swarm-worker`.

4.5.3. Дослідження відмовостійкості системи

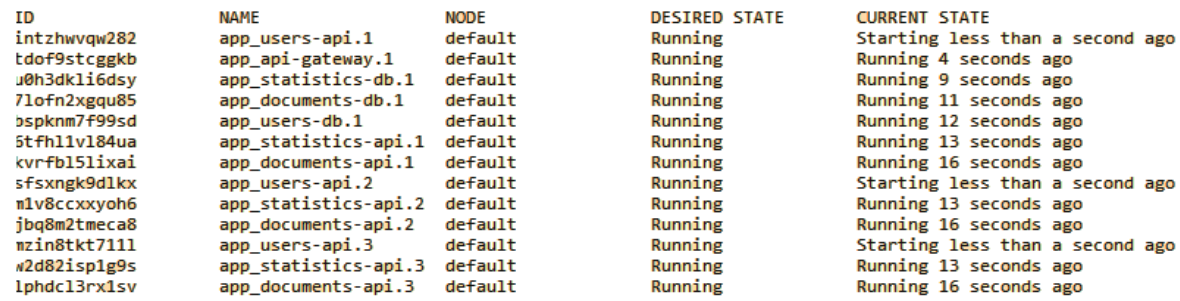
Відмовостійкість кластеру гарантується самим Docker. Це досягається в тому числі за рахунок того, що в кластері можуть одночасно працювати декілька керуючих нод, які можуть в будь-який момент замінити лідера, що вийшов з ладу. Використовується так званий алгоритм підтримки розподіленого консенсусу – Raft.

Raft реалізується поверх кластера одноманітних слабо зв'язкових нод, на кожній з яких працює машина станів, таким чином, кожна нода гарантовано стає в згоду з іншими нодами.

Відмовостійкість сервісів, а саме екземплярів контейнерів даних сервісів, гарантується за допомогою реплікації: контейнери одного сервіса можуть розміщуватись на декількох нодах.

Наприклад, відключимо усі ноди крім головної (*default*) за допомогою команди `docker-machine.exe stop [node_name]`. Після зупинки нод, усі контейнери мігрували на ноду що залишилась – *default*, що видно на рис. 4.10.

Тривалість міграції на доступну ноду: 12.67 секунд.



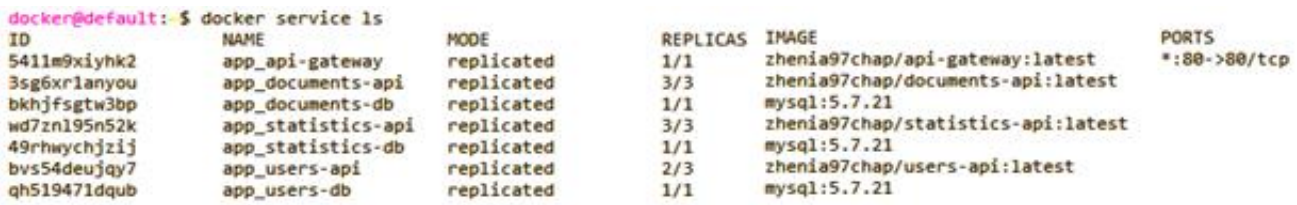
ID	NAME	NODE	DESIRED STATE	CURRENT STATE
intzhwvqw282	app_users-api.1	default	Running	Starting less than a second ago
tdof9stcggkb	app_api-gateway.1	default	Running	Running 4 seconds ago
u0h3dkli6dsy	app_statistics-db.1	default	Running	Running 9 seconds ago
7lofn2xgqu85	app_documents-db.1	default	Running	Running 11 seconds ago
bspknm7f99sd	app_users-db.1	default	Running	Running 12 seconds ago
5tfhl1vl84ua	app_statistics-api.1	default	Running	Running 13 seconds ago
kvrfbl5lixai	app_documents-api.1	default	Running	Running 16 seconds ago
sfsxngk9dlkx	app_users-api.2	default	Running	Starting less than a second ago
m1v8ccxxyoh6	app_statistics-api.2	default	Running	Running 13 seconds ago
jbq8m2tmeca8	app_documents-api.2	default	Running	Running 16 seconds ago
mzin8tkt711l	app_users-api.3	default	Running	Starting less than a second ago
w2d82isp1g9s	app_statistics-api.3	default	Running	Running 13 seconds ago
lphdcl3rx1sv	app_documents-api.3	default	Running	Running 16 seconds ago

Рис. 4.10. Забезпечення відмовостійкості сервісів

Для перевірки відмовостійкості окремо взятого контейнера будемо використовувати симуляцію нештатної зупинки контейнера. Для зупинки контейнера використовується команда *stop*, наприклад: `docker stop 783ee96f6500`, де, `783ee96f6500` – унікальний ідентифікатор контейнера.

На рис. 4.11 видно, що сервіс *users-api*, включає тільки два контейнера.

Тривалість запуску нового контейнера: 4.3 с.



```
docker@default: $ docker service ls
ID              NAME                MODE          REPLICAS  IMAGE                                  PORTS
5411m9xiyhk2    app_api-gateway     replicated    1/1       zhenia97chap/api-gateway:latest        *:80->80/tcp
3sg6xr1anyou    app_documents-api   replicated    3/3       zhenia97chap/documents-api:latest
bkhjfsgtw3bp    app_documents-db    replicated    1/1       mysql:5.7.21
wd7znl95n52k    app_statistics-api  replicated    3/3       zhenia97chap/statistics-api:latest
49rhwychjzij    app_statistics-db   replicated    1/1       mysql:5.7.21
bvs54deujqy7    app_users-api       replicated    2/3       zhenia97chap/users-api:latest
qh519471dqub    app_users-db        replicated    1/1       mysql:5.7.21
```

Рис. 4.11. Аварійна зупинка контейнера

4.5.4. Аналіз затримок у міжпроцесорній взаємодії

Одним із недоліків мікросервісної архітектури є мережеві затримки при взаємодії сервісів один з одним.

Для мінімізації даного недоліку найчастіше використовуються принцип за яким контейнери, які часто взаємодіють між собою, розміщуються в одній мережі або у географічно близьких дата-центрах. Це дає змогу зменшити час очікування відповіді від іншого сервіса при синхронній взаємодії між сервісами. У випадку використання асинхронної моделі взаємодії – сервіс не чекає відповіді і продовжує виконання інструкцій.

Проведемо дослідження тривалості затримки відповіді для REST-ресурсу: `api/v1/users/[user_id]`.

У даному випадку запит перенаправляється на сервіс *users-api*, який взаємодіє і отримує дані із сервісів *documents-api* і *statistics-api*.

У табл. 4.2 приведено аналіз часу очікування відповіді від додатку. Загалом час очікування склав 1,09 с. При чому, найшвидше запит пройшов через сервіс *api gateway*, а найдовше – *users-api*.

Отже, можна зробити висновок, що на маршрутизацію запиту витрачається найменше часу.

Таблиця 4.2

Аналіз часу очікування відповіді

Сервіс	Тривалість, секунди
Api Gateway	0,19
Users-api	0,38
Documents-api	0,22
Statistics-api	0,3
Усього	1,09

4.6. Моніторинг системи

Моніторинг додатків і серверів додатків – важливий компонент кожної інфраструктури. Існує потреба у постійному моніторингу стану контейнерів, серверів, завантаження центрального процесора, споживання пам'яті, дискову утилізацію і т.д. Також є додаткова необхідність у сповіщені розробників, якщо у сервера закінчується доступна пам'ять або додаток перестає відповідати на запити, що дозволить запобігти проблемам, які можуть у перспективі призвести до відмови сервера.

Prometheus – система моніторингу різних систем і сервісів, за допомогою якої системні адміністратори можуть збирати інформацію про поточні параметри, налаштовувати конфігурацію для отримання інших повідомлень (наприклад, попередження, коли завантаження CPU перейшло за граничну межу).

Використовуючи Prometheus, опишемо необхідні групи метрик:

- метрики хост машини;
- метрики контейнерів;
- метрики api-gateway (єдина точка входу у додаток).

Вміст конфігураційного файлу Prometheus наведено нижче. Блок *targets* описує звідки брати метрики, у даному випадку це контейнери експортерів метрик. Інструкція *scrape_interval* вказує з яким інтервалом опитувати експортерів.

```
global:
  scrape_interval: 5s
  evaluation_interval: 5s
  external_labels:
    monitor: 'prometheus-grafana-exporter'

scrape_configs:
  - job_name: 'node-exporter'
    scrape_interval: 10s
    static_configs:
      - targets: ['node-exporter:9100']

  - job_name: 'nginx-exporter'
    scrape_interval: 10s
    static_configs:
      - targets: ['nginx-exporter:9113']

  - job_name: 'containers-exporter'
    scrape_interval: 10s
    static_configs:
      - targets: ['cadvisor:8080']
```

Експортер – частина програмного забезпечення, яка отримує існуючі метрики від сторонньої системи і експортує їх в формат, зрозумілий для сервера Prometheus.

На рис. 4.12 зображено графік використання CPU хост машини default. Із графіку видно, що CPU навантаження залишається стабільним на рівні ~12%. Це пояснюється тим, що система більшу частину часу ненавантажена, так як немає трафіку користувачів.

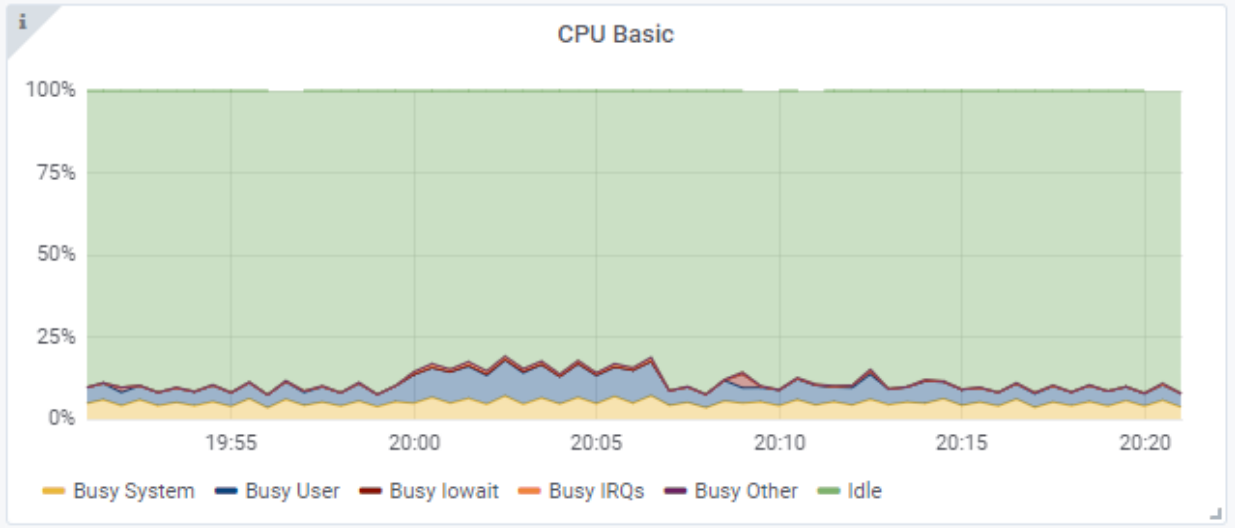


Рис. 4.12. Графік використання CPU хост машиною

На рис. 4.13 зображено графік використання CPU кожним запущеним контейнером.

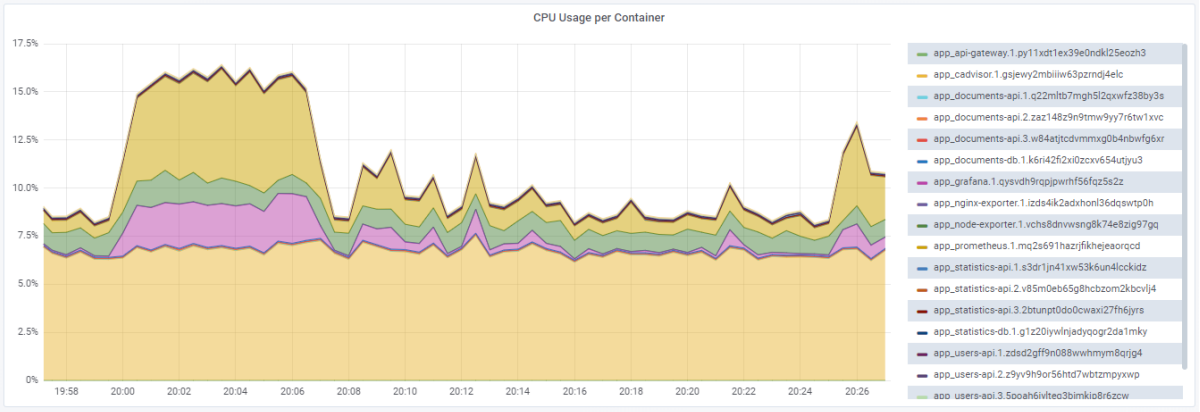


Рис. 4.13. Графік використання CPU контейнерами

Використовуючи даний графік можна визначити, який контейнер найбільше навантажує процесор і прийняти міри: наприклад, додати декілька екземплярів сервісу на інший хост.

Запит на отримання метрик для рис. 4.13 наведений нижче:

```
sum(rate(container_cpu_usage_seconds_total{name=~".+"}[$interval])) by (name) * 100
```

На рис. 4.14 зображено таблицю використання RAM контейнерами із якої видно що система найбільше навантажена моніторинговими сервісами, які сканують метрики хоста, а інші сервіси простоюють так як мережевого трафіку немає.

Запит для отримання метрик використання RAM контейнерами наведено нижче:

```
container_memory_usage_bytes{name=~".+"}
```

Metric	Current
app_statistics-api.2.v85m0eb65g8hcbzom2kbcvlj4	12.60 MB
app_statistics-api.1.s3dr1jn41xw53k6un4lcckidz	13.00 MB
app_prometheus.1.mq2s691hazrjfikhejeaorqcd	94.88 MB
app_node-exporter.1.vchs8dnvwsng8k74e8zig97gq	12.75 MB
app_nginx-exporter.1.izds4ik2adxhonl36dqswtp0h	6.97 MB
app_grafana.1.qysvdh9rqpjpwrhf56fqz5s2z	48.51 MB
app_documents-db.1.k6ri42fi2xi0zcxv654utjyu3	77.32 MB
app_documents-api.3.w84atjtcdvmmxg0b4nbwfg6xr	14.75 MB
app_documents-api.2.zaz148z9n9tmw9yy7r6tw1xvc	12.72 MB
app_documents-api.1.q22mltb7mgh5l2qxwfz38by3s	13.15 MB
app_cadvisor.1.gsjewy2mbiiiw63pzrndj4elc	49.65 MB
app_api-gateway.1.py11xdt1ex39e0ndkl25eozh3	2.34 MB

Рис. 4.14. Використання RAM контейнерами

На рис. 4.15 зображено графік із інформацією про кількість прийнятих з’єднань сервісом *api-gateway* за визначений період часу.

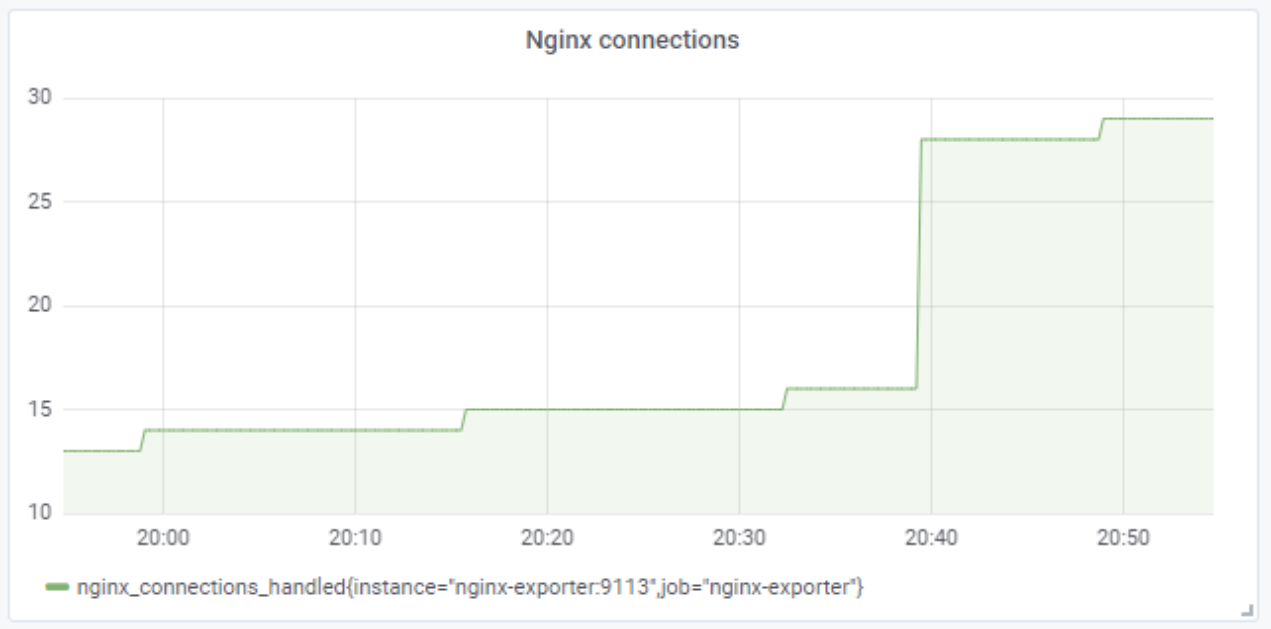


Рис. 4.15. Кількість прийнятих з’єднань nginx

Висновок

Програмне забезпечення Docker Swarm дозволяє створювати кластерну інфраструктуру, яка масштабованою та стійкою до відмов.

Було створено чотири мікросервіса і організовано взаємодію між ними при цьому було застосовано стандартизований підхід на основі REST-API та JSON API для взаємодії між сервісами.

Використовуючи платформу DockerHub було реалізовано концепцію безперервної доставки, що значно спростило і пришвидшило розробку програмного забезпечення.

Було проаналізовано та досліджено відмовостійкість системи і визначено що кластер відновлює свій стан після відмови деякого вузла.

За допомогою Prometheus було створено моніторинг системи, що дає зможу своєчасно виявляти потенційні проблеми у роботі серверу або сервісів.

ВИСНОВКИ

У ході виконання дипломного проекту було реалізовано і спроектовано систему контейнеризації мікросервісів.

Було проаналізовано складові та компоненти веб-інфраструктури, принципи взаємодії клієнт-сервер, визначено базові технології і служби за допомогою яких організовано роботу веб-додатків.

Протокол передачі гіпертексту HTTP – це прикладний протокол для передачі гіпертекстових документів, таких як HTML, створений для зв'язку між веб-браузерами і веб-серверами та може використовуватися і для інших цілей. Даний протокол реалізує класичну клієнт-серверну модель, коли клієнт відкриває з'єднання для створення запиту, а потім чекає відповіді.

HTTP – це протокол без збереження стану, тобто сервер не зберігає ніяких даних (стан) між двома парами «запит-відповідь». Незважаючи на те, що HTTP заснований на TCP / IP, він також може використовувати будь-який інший протокол транспортного рівня з гарантованою доставкою.

Було проведено аналіз і дослідження принципів побудови мікросервісної архітектури. Було зроблено висновок, що архітектурний стиль мікросервісів – це підхід, при якому єдиний додаток будується як набір невеликих сервісів, кожен з яких працює у власному процесі і взаємодіє з іншими, використовуючи легковагі механізми зв’язку, як правило HTTP.

Дані сервіси побудовані навколо бізнес-потреб і розгортаються незалежно з використанням повністю автоматизованої середовища. Самі по собі сервіси можуть бути написані на різних мовах і використовувати різні технології зберігання даних.

Було проведено аналіз і порівняльну характеристику мікросервісного підходу із іншими шаблонами проектування, такими як: монолітна архітектура та безсерверна архітектура.

Було визначено що мікросервісний підхід має ряд переваг, такі як: модульність, незалежність розгортання та розробка, автономність та ізоляція відмов.

Було проаналізовано недоліки мікросервісного підходу, а саме: складність стандартизації мікросервісної взаємодії, необхідність версіонування сервісів для забезпечення сумісності служб.

Проаналізовано основні види міжпроцесорної взаємодії: синхронну та асинхронну.

Було зроблено висновок, що якщо додаток відносно невеликий, то можна обійтися підтримкою REST-запитів, тобто на базі синхронної взаємодії. Це значно спрощує архітектуру додатку в цілому, але призводить до істотних витрат на передачу інформації. При досить складному додатку без створення менеджера не обійтися.

Проаналізувавши обидва підходи, можна зробити висновок, що для великих додатків, які складаються із десятків сервісів, необхідно реалізувати менеджер повідомлень, щоб гарантувати швидку доставку та уникнути блокування від очікування відповіді.

Менеджер повинен приймати асинхронний запит від одного сервісу і передавати його іншому. Він може бути реалізований на сокетах, веб-сокетах, або на будь-якій іншій технології. Запити, як правило, зберігаються у чергах. При такому підході з'являється простий інструмент моніторингу взаємодії сервісів між собою.

Було досліджено основні шаблони при проектуванні мікросервісів, а саме шаблон ізоляції відмов, шаблон автономності службт, шаблон модульності сервісу.

Проведено аналіз базових підходів у розгортанні додатків, а саме із використанням апаратної віртуалізації і програмної контейнеризації, було проаналізовано основі принципи роботи та переваги і недоліки кожної із стратегій.

Проведено аналіз концепції безперервної доставки.

Було досліджено принципи керування кластером сервісів, а саме кластеризація – дублювання сервісів і створення єдиного кластеру та оркестрація – керування кластером, масштабування, моніторинг відмов та збоїв у системі.

У якості програмного забезпечення для контейнеризації було вибрано Docker, як сучасний і широко застосований продукт для керування контейнерами і образами.

Було проаналізовано основні складові Docker, а саме: образи, мережа для взаємодії між контейнерами, томи для зберігання персистентних даних та контейнери. Проведено аналіз рушія Docker – Docker-демона та Docker-клієнта.

За допомогою системи кластеризації Docker Swarm було побудовано відмовостійкий і масштабований кластер мікросервісів.

Кожен сервіс було продубльовано у кількість трьох контейнерних реплік, що дало змогу рівномірно розподілити навантаження і досягти автономності сервісу – у разі відмови одного із контейнерів, оркестратор Docker Swarm відновить потрібну кількість реплік.

Було проведено випробування перевірки відмостійкості кластеру шляхом виведення з ладу одного із хостів. Було встановлено, що система відновила свій стан у швидкий термін, перенаправивши усі сервіси та контейнери на інший доступний хост.

Було проведено випробування перевірки відмостійкості кластеру шляхом виведення з ладу одного із контейнерів сервісу. Було встановлено, що оркестратор Docker Swarm заново створив контейнер, відновивши стан системи і необхідну кількість реплік сервісу.

Було реалізовано концепцію безперервної доставки, що дало змогу швидко розгортати потрібні сервіси.

Використовуючи систему моніторингу Prometheus, було створено оточення для нагляду і контролю над системою. Створені графіки, наприклад, навантаження CPU, RAM, і діаграми дають змогу оцінити навантаження на сервер і вчасно прийняти міри.

Можна зробити висновок що недоліками контейнерів є аспект конфіденційності і безпеки, адже використовується програмна віртуалізація. Перевагою контейнерів є менші ресурсні витрати як дискового простору, так і RAM, CPU у порівнянні із віртуальними машинами.

Отже, віртуалізація підходить найкраще для ізольованих систем, для яких питання безпеки є надважливим.

Контейнеризація підходить для більшості випадків, адже можливо на мінімальній кількості серверів запустити значну кількість додатків завдяки контейнерам.

Таким чином, створена кластерна відмовостійка, масштабована інфраструктура і реалізовані такі додаткові компоненти, як: моніторинг, безперервна доставка, модульність сервісів, стандартизація формату обміну, дають змогу на базі створеного проекту реалізувати комерційний продукт із мінімальними час на проектування інфраструктури, потрібно лише реалізувати бізнес-логіку сервісів.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Микросервисные паттерны проектирования [Electronic resource]. Access mode: https://habr.com/ru/company/piter/blog/275633/ (lastaccess:22.11.19).

2. Розгортання монолітного додатку [Electronic resource]. Access mode: https://aws.amazon.com/ru/getting-started/container-microservices-tutorial/module-two/ (lastaccess:20.11.19).

3. Ньюмен С. Создание микросервисов. — СПб.: Питер, 2016. — 304 с.: ил. — (Серия «Бестселлеры O'Reilly»).

4. Шаблон отсеков [Electronic resource]. Access mode: https://docs.microsoft.com/ru-ru/azure/architecture/patterns/bulkhead (lastaccess:07.01.20).

5. Шаблон подавления [Electronic resource]. Access mode: https://docs.microsoft.com/ru-ru/azure/architecture/patterns/strangler (lastaccess:08.01.20).

6. Шаблон расширения [Electronic resource]. Access mode: https://docs.microsoft.com/ru-ru/azure/architecture/patterns/sidecar (lastaccess:08.01.20).

7. Взаимодействие в архитектуре микрослужб [Electronic resource]. Access mode: https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture (lastaccess:14.01.20).

8. Хорсдал К. Микросервисы на платформе .NET. — СПб.: Питер, 2018. — 352 с.: ил. — (Серия «Для профессионалов»).

9. Microservice Architecture and Design Patterns for Microservices [Electronic resource]. Access mode: https://dzone.com/articles/microservice-architecture-and-design-patterns-for (lastaccess:10.01.20).

10. Экосистема Docker: обнаружение сервисов (Service Discovery) и распределённые хранилища конфигураций (Distributed Configuration Stores) [Electronic resource]. Access mode: https://www.digitalocean.com/community/tutorials/docker-service-discovery-distributed-configuration-stores-ru (lastaccess:10.01.20).

11. Сравнительный анализ форматов обмена данными, используемых в приложениях с клиент-серверной архитектурой [Electronic resource]. Access mode: https://www.fundamental-research.ru/ru/article/view?id=38464 (lastaccess:12.01.20).

12. Непрерывная интеграция [Electronic resource]. Access mode: https://ru.wikipedia.org/wiki/Непрерывная_интеграция (lastaccess:08.01.20).

13. Выбор стратегии деплоя микросервисов [Electronic resource]. Access mode: https://bool.dev/blog/detail/vybor-strategii-deploya-mikroservisov (lastaccess:02.01.20).

14. Ричардсон Крис Микросервисы. Паттерны разработки и рефакторинга. — СПб.: Питер, 2019. — 544 с.: ил. — (Серия «Библиотека программиста»).

15. Кластер серверов микросервисов [Electronic resource]. Access mode: https://www.stekspb.ru/autsorsing-it-infrastruktury/it-glossary/server-cluster/ (lastaccess:22.01.20).

16. Основы мониторинга и сбора метрик [Electronic resource]. Access mode: https://www.8host.com/blog/osnovy-monitoringa-i-sbora-metrik/ (lastaccess:10.01.20).

17. Docker и технология контейнеров Linux [Electronic resource]. Access mode: https://vps.ua/blog/docker-and-linux-containers/ (lastaccess:13.01.20).

18. Технология контейнеризации [Electronic resource]. Access mode: https://www.cloud4y.ru/about/news/obzor-tekhnologii-konteynerizatsii/ (lastaccess:20.01.20).

19. Моуэт Э. Использование Docker / пер. с англ. А. В. Снастина; науч. ред. А. А. Маркелов. – М.: ДМК Пресс, 2017. – 354 с.: ил.

20. About images, containers, and storage drivers [Electronic resource]. Access mode: https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers (lastaccess:22.01.20).

21. Основы Kubernetes [Electronic resource]. Access mode: https://habr.com/ru/post/258443/ (lastaccess:21.01.20).

22. Взаимодействие docker контейнеров [Electronic resource]. Access mode: https://dotsandbrackets.com/communication-between-docker-containers-ru/ (lastaccess:21.01.20).

23. Экосистема Docker: сетевое взаимодействие [Electronic resource]. Access mode: https://www.digitalocean.com/community/tutorials/docker-ru-992094e0-5e33-49a5-b30f-f9bfa371aeab (lastaccess:21.01.20).

Додаток А

Код конфігураційного файлу кластера Docker Swarm

```
version: "3.7"

services:
  api-gateway:
    image: zhenia97chap/api-gateway:latest
    deploy:
      replicas: 1
    ports:
      - 80:80
    depends_on:
      - users-api
      - documents-api
      - statistics-api

  users-api:
    image: zhenia97chap/users-api:latest
    deploy:
      replicas: 3
    depends_on:
      - users-db

  documents-api:
    image: zhenia97chap/documents-api:latest
    deploy:
      replicas: 3
    depends_on:
      - documents-db

  statistics-api:
    image: zhenia97chap/statistics-api:latest
    deploy:
      replicas: 3
    depends_on:
      - statistics-db

  users-db:
    image: mysql:5.7.21
    deploy:
      replicas: 1
    environment:
      MYSQL_USER: root
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: users-api
    volumes:
```

```
    - db_users_data:/var/lib/mysql

  documents-db:
    image: mysql:5.7.21
    deploy:
      replicas: 1
    environment:
      MYSQL_USER: root
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: documents-api
    volumes:
      - db_documents_data:/var/lib/mysql

  statistics-db:
    image: mysql:5.7.21
    deploy:
      replicas: 1
    environment:
      MYSQL_USER: root
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: statistics-api
    volumes:
      - db_statistics_data:/var/lib/mysql

  prometheus:
    image: prom/prometheus:v2.15.2
    deploy:
      placement:
        constraints:
          - node.role == manager
    volumes:
      - ./prometheus/:/etc/prometheus/
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'
      - '--storage.tsdb.retention=200h'
      - '--web.enable-lifecycle'
    ports:
      - 9090:9090

  node-exporter:
    image: prom/node-exporter:v0.18.1
    user: root
    volumes:
      - /proc:/host/proc:ro
```

```
      - /sys:/host/sys:ro
      - /:/rootfs:ro
    command:
      - '--path.procfs=/host/proc'
      - '--path.sysfs=/host/sys'
      - '--collector.filesystem.ignored-mount-points=^/(sys|proc|dev|host|etc)($$|/)'

  nginx-exporter:
    image: nginx/nginx-prometheus-exporter:0.5.0
    environment:
      - SCRAPE_URI=http://192.168.99.100/nginx_status
      - TELEMETRY_PATH=/metrics
      - NGINX_RETRIES=10
    logging:
      driver: "json-file"
      options:
        max-size: "5m"

  cadvisor:
    image: google/cadvisor:latest
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro

  grafana:
    image: grafana/grafana:latest
    volumes:
      - grafana_data:/var/lib/grafana
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=admin
      - GF_USERS_ALLOW_SIGN_UP=false
    ports:
      - 3000:3000

volumes:
  db_users_data:
    driver: local
  db_documents_data:
    driver: local
  db_statistics_data:
    driver: local
  prometheus_data:
    driver: local
  grafana_data:
    driver: local
```

Додаток Б

Вміст Dockerfile мікросервісів users-api, documents-api, statistics-api

```
FROM php:7.2-fpm

RUN apt-get update \
    && apt-get -y install git \
    && apt-get -y install zip \
    && apt-get -y install procps \
    && apt-get -y install htop \
    && apt-get -y install nano \
    && apt-get -y install supervisor \
    && apt-get -y install net-tools \
    && apt-get -y install nginx


RUN docker-php-ext-install pdo_mysql


COPY . /var/www
COPY docker/supervisor/conf.d /etc/supervisor/conf.d/
COPY docker/nginx/conf.d/default.conf /etc/nginx/conf.d


RUN chown -R www-data:www-data /var/www
RUN rm /etc/nginx/sites-available/default /etc/nginx/sites-enabled/default


WORKDIR /var/www


RUN curl -sS https://getcomposer.org/installer | php -- --install-
dir=/usr/local/bin --filename=composer
RUN composer install -n --prefer-dist --ignore-platform-reqs


RUN mv .env.example .env
RUN php artisan key:generate


CMD ["/usr/bin/supervisord", "-n"]
```

Додаток В

Конфігурація веб-сервера сервісу api-gateway

```
server {
    listen 80;
    index index.php index.html;
    root /var/www/public;

    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;

    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_index index.php;

    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME ${document_root}/index.php;
    fastcgi_param PATH_INFO $fastcgi_path_info;

    location / {
        return 200 'App is alive';
        add_header Content-Type text/plain;
    }

    location /api/v1/users {
        rewrite ^(.*)/$ $1 permanent;
        fastcgi_pass users-api:9000;
    }

    location /api/v1/documents {
        rewrite ^(.*)/$ $1 permanent;
        fastcgi_pass documents-api:9000;
    }

    location /api/v1/statistics {
        rewrite ^(.*)/$ $1 permanent;
        fastcgi_pass statistics-api:9000;
    }
}
```